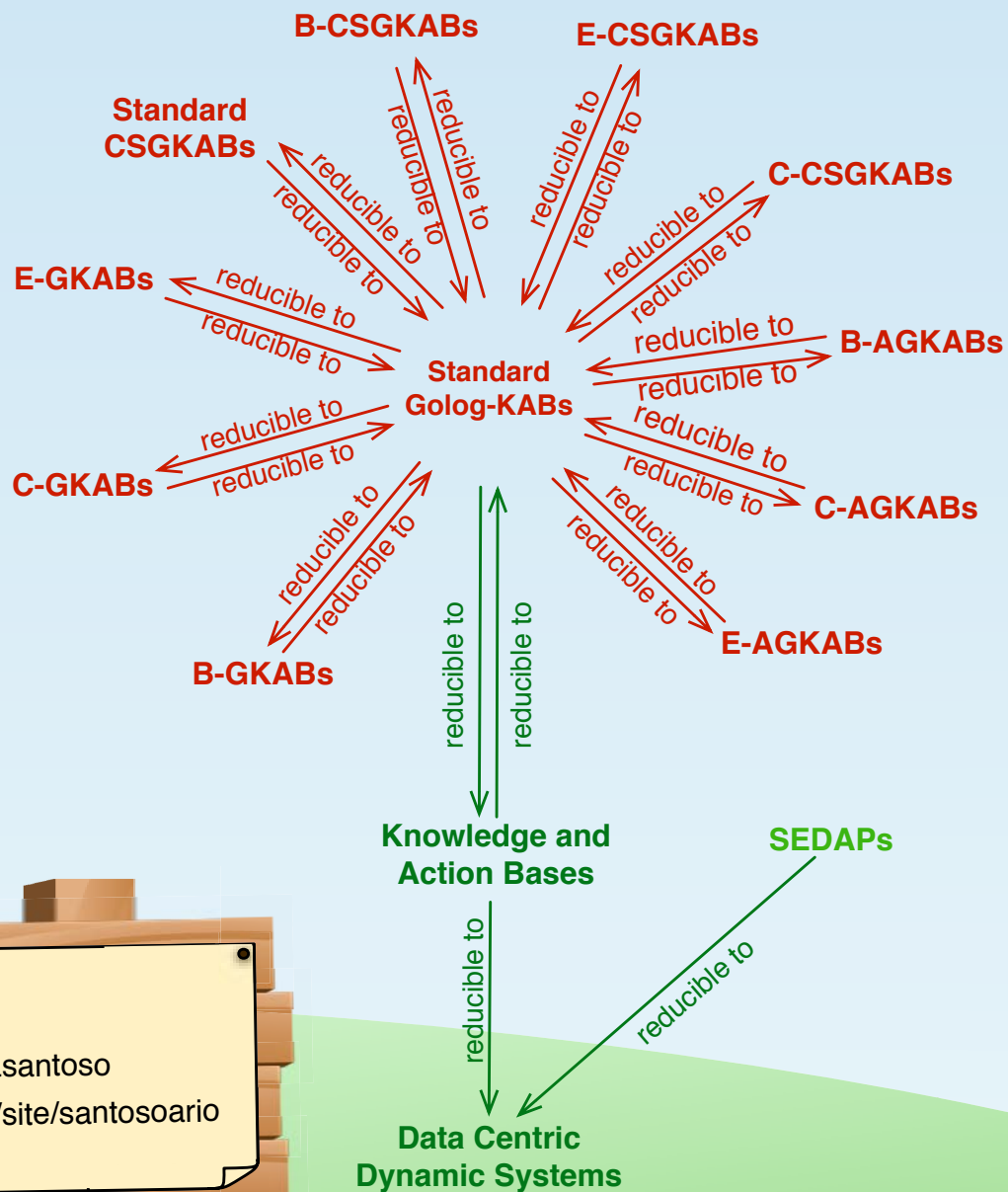


Verification of Data-aware Business Processes in the Presence of Ontologies



arXiv:1612.05456v1 [cs.LO] 16 Dec 2016

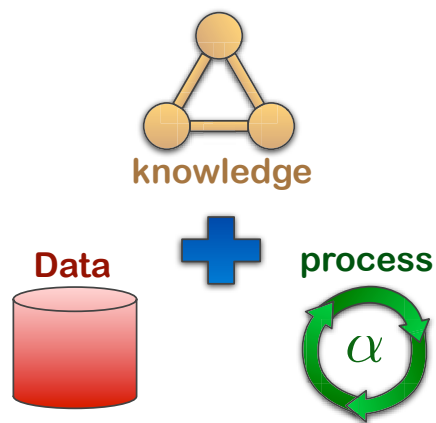


Ario Santoso

<http://www.inf.unibz.it/~asantoso>

<https://sites.google.com/site/santosoario>

VERIFICATION OF DATA-AWARE BUSINESS PROCESSES IN THE PRESENCE OF ONTOLOGIES



ARIO SANTOSO
santoso@inf.unibz.it
santoso.ario@gmail.com
<http://www.inf.unibz.it/~asantoso>
<https://sites.google.com/site/santosoario>

VERIFICATION OF DATA-AWARE BUSINESS PROCESSES IN THE PRESENCE OF ONTOLOGIES

ARIO SANTOSO



European PhD Program in Computational Logic (EPCL)

FIRST INSTITUTION:

Facoltà di Scienze e Tecnologie Informatiche
Libera Università di Bolzano
Italy

SECOND INSTITUTION:

Fakultät Informatik
Technische Universität Dresden
Germany

April 2016

Ario Santoso

Verification of Data-aware Business Processes in the Presence of Ontologies

PhD Dissertation, © April 2016

SUPERVISOR:

Diego Calvanese - <http://www.inf.unibz.it/~calvanese>

Libera Università di Bolzano (Bolzano - Italy)

CO-SUPERVISOR:

Marco Montali - <http://www.inf.unibz.it/~montali>

Libera Università di Bolzano (Bolzano - Italy)

EXTERNAL SUPERVISOR:

Franz Baader - <http://lat.inf.tu-dresden.de/~baader>

Technische Universität Dresden (Dresden - Germany)

REVIEWERS:

Yves Lespérance - <http://www.cse.yorku.ca/~lesperan>

York University (Toronto - Canada)

Sebastian Sardiña - <http://www1.rmit.edu.au/staff/sebastian-sardina>

Royal Melbourne Institute of Technology - RMIT University (Melbourne - Australia)

PHD DEFENSE COMMITTEE:

Sven Helmer - <http://www.inf.unibz.it/~shelmer>

Libera Università di Bolzano (Bolzano - Italy)

Steffen Hölldobler - <http://www.computational-logic.org/~sh>

Technische Universität Dresden (Dresden - Germany)

Gerhard Lakemeyer - <http://www-i5.informatik.rwth-aachen.de/~gerhard>

RWTH Aachen University (Aachen - Germany)

To You,
who illuminates my paths.

To all of my teachers, lecturers, and professors,
who have taught and guided me until I can reach this far.

To everyone who has influenced my journey up to this moment,
good or bad, it shapes me to what I am now.

To my living caffeine,
who always be there and supports me.

ABSTRACT

The meet up between data, processes and structural knowledge in modeling complex enterprise systems is a challenging task that has led to the study of combining formalisms from knowledge representation, database theory, and process management. Moreover, to ensure system correctness, formal verification also comes into play as a promising approach that offers well-established techniques. In line with this, significant results have been obtained within the research on data-aware business processes, which studies the marriage between static and dynamic aspects of a system within a unified framework. However, several limitations are still present. Various formalisms for data-aware processes that have been studied typically use a simple mechanism for specifying the system dynamics. The majority of works also assume a rather simple treatment of inconsistency (i.e., reject inconsistent system states). Many researches in this area that consider structural domain knowledge typically also assume that such knowledge remains fixed along the system evolution (context-independent), and this might be too restrictive. Moreover, the information model of data-aware processes sometimes relies on relatively simple structures. This situation might cause an abstraction gap between the high-level conceptual view that business stakeholders have, and the low-level representation of information. When it comes to verification, taking into account all of the aspects above makes the problem more challenging.

In this thesis, we investigate the verification of data-aware processes in the presence of ontologies while at the same time addressing all limitations above. Specifically, we provide the following contributions: (1) We propose a formal framework called Golog-KABs (GKABs), by leveraging on the state of the art formalisms for data-aware processes equipped with ontologies. GKABs enable us to specify semantically-rich data-aware business processes, where the system dynamics are specified using a high-level action language inspired by the Golog programming language. (2) We propose a parametric execution semantics for GKABs that is able to elegantly accommodate a plethora of inconsistency-aware semantics based on the well-known notion of repair, and this leads us to consider several variants of inconsistency-aware GKABs. (3) We enhance GKABs towards context-sensitive GKABs that take into account the contextual information during the system evolution. (4) We marry these two settings and introduce inconsistency-aware context-sensitive GKABs. (5) We introduce the so-called Alternating-GKABs that allow for a more fine-grained analysis over the evolution of inconsistency-aware context-sensitive systems. (6) In addition to GKABs, we introduce a novel framework called Semantically-Enhanced Data-Aware Processes (SEDAPs) that, by utilizing ontologies, enable us to have a high-level conceptual view over the evolution of the underlying system. We provide not only theoretical results, but have also implemented this concept of SEDAPs.

We also provide numerous reductions for the verification of sophisticated first-order temporal properties over all of the settings above, and show that verification can be addressed using existing techniques developed for Data-Centric Dynamic Systems (which is a well-established data-aware processes framework), under suitable boundedness assumptions for the number of objects freshly introduced in the system while it evolves. Notably, all proposed GKAB extensions have no negative impact on computational complexity.

ACKNOWLEDGMENTS

I would like to express my gratitude to Prof. Diego Calvanese, who has been my great supervisor even since I did my master thesis. Also thanks to Marco Montali who has become my great co-supervisor. Thanks to both of them for their unbounded help, guidance and kindness. Their greatness might even be inexpressible in words.

Sincere thanks to the Free University of Bozen-Bolzano (FUB) and the KRDB Research Centre for Knowledge and Data, which have provided me with plenty of support that I can not mention one by one. Thanks to Viviana Foscarin, the student secretariat personnel in the Faculty of Computer Science at FUB, for all of the support related to any administrative issues. Also thanks to Technische Universität Dresden (TUD) and European PhD Program in Computational Logic (EPCL) for all of the support related to my study within the EPCL program as well as during my research visit in TUD.

I would also like to express my gratitude to Prof. Franz Baader who has been my supervisor within the EPCL Program and also for hosting me in his group at Lehrstuhl für Automatentheorie of TUD. I would also like to thank all of the colleagues in Lehrstuhl für Automatentheorie of TUD for the hospitality during my stay. Especially, thanks to Anni Yasmin Turhan for all the fascinating discussions and help. Thanks to Stephan Böhme for all the interesting discussions regarding the Rösi project. Thanks to İsmail İlkan Ceylan who has introduced me into the works on context and also thanks for our fruitful discussions and collaborations that have led to nice outcomes related to the work on context. Thanks to Benjamin Zarriß for all the nice discussions, especially related to Golog, which has become an important part in this thesis. Thanks to Mrs. Achtruth who gave me enormous helps on administrative matters during my stay in TUD. Last but not least, thanks to all of my colleagues in TUD who gave me great experiences during my stay there.

I would also like to express my gratitude to Yves Lespérance and Sebastian Sardiña who have become great reviewers for this thesis. Thank you so much for the time that you have spend in reading this thesis and of course for all of your fruitful comments, feedback and suggestions that really improve the thesis.

I would also like to thank Prof. Steffen Hölldobler and Christoph Wernhard who have coordinated the EPCL program. Also thanks to Julia Kopenhagen and Sylvia Wünsch for all of the support related to administrative aspects within EPCL.

Also, I would like to express my gratitude to Babak Bagheri Hariri, Giuseppe De Giacomo, Evgeny Kharlamov, Domenico Lembo, Dmitry Solomakhin, and Dmitriy Zheleznyakov for all of the collaborations that have led to some of the results in this thesis.

I would also like to say thanks to all of my colleagues in KRDB whom I cannot mention one by one. I would also like to express my thanks to all of the people who gave me wonderful times during my stay in Dresden, especially the people in Formid e.V., and also all of my friends there whom I can not elaborate one by one. Also

thanks to all of my friends in Bolzano and Trento for giving me a great time during my stay in Bolzano.

Also deep thanks to Eliza Margaretha and Annisa Ihsani for some writing enhancements and writing discussions concerning this thesis. Additionally, thanks to Arya and Alida Widiанти for some discussions regarding business scenarios for the running examples on this thesis.

Last but not least, special thanks to my living caffeine for several writing enhancements, and most prominently for her endless support that always strengthen me in various difficult moments. In the end, I would like to say thanks to everyone who has supported me in this journey, though their names might not be in this page, they are always in my mind and especially in my heart.

Bolzano - Italy, April 2016
Ario Santoso

CONTENTS

1	INTRODUCTION	1
1.1	The Meet Up Between Data, Processes and Knowledge	3
1.2	Research Challenges	7
1.3	Contributions	10
1.4	List of Publications	19
1.5	Organization of the Thesis	22
2	PRELIMINARIES	25
2.1	Scenario for the Running Examples	25
2.2	Some Basic Notions and Notations Convention	25
2.3	Relational Databases	26
2.4	<i>DL-Lite</i> Knowledge Bases	27
2.5	Query Answering	32
2.6	History Preserving μ -Calculus	39
2.7	Data Centric Dynamic Systems (DCDSs)	41
3	KNOWLEDGE AND ACTION BASES (KABs)	53
3.1	KABs Formalism	53
3.2	KABs Standard Execution Semantics	57
3.3	Verification of KABs	61
3.4	Discussion: Weakly Acyclic KABs	72
4	GOLOG-KABs (GKABs)	73
4.1	GKABs Formalism	73
4.2	GKABs Standard Execution Semantics	76
4.3	Capturing KABs within Standard GKABs	85
4.4	Verification of Standard GKABs (S-GKABs)	87
4.5	Discussion	121
5	INCONSISTENCY-AWARE GOLOG-KABs (I-GKABs)	127
5.1	Inconsistency Management in DL KBs	128
5.2	Inconsistency-Aware Semantics for GKAB.	131
5.3	Compilation of Inconsistency Management	135
5.4	Back From Standard to Inconsistency-aware GKABs	165
5.5	Discussion: Extended Inconsistency-Aware Golog-KABs	168
6	EMBRACING CONTEXTS INTO GOLOG-KABs	171
6.1	Context Formalization	172
6.2	Contextualizing Knowledge Bases	175
6.3	Contextualizing Golog-Program	177
6.4	Context-Sensitive Golog-KABs (CSGKABs)	177
6.5	Verifying Temporal Properties over Standard CSGKAB	186
6.6	Capturing Standard GKABs within Standard CSGKABs	200
6.7	Discussion	203
7	INCONSISTENCY-AWARE CONTEXT-SENSITIVE GKABs	205
7.1	The Inconsistency-aware Context-sensitive Execution Semantics	206

7.2	From Inconsistency-aware Context-sensitive to Standard GKABs . . .	210
7.3	From Standard to Inconsistency-aware Context-sensitive GKABs . . .	227
8	ALTERNATING GOLOG-KABS (AGKABS)	233
8.1	AGKABS Formalism and Execution Semantics	234
8.2	Verification of AGKABS	238
8.3	Emulating Standard GKABs in Alternating GKABs	267
8.4	Discussion: Connection between Inconsistency-aware Context-sensitive GKABs and AGKABS	269
9	SEMANTICALLY-ENHANCED DATA-AWARE PROCESSES (SEDAPs)	271
9.1	Formalizing SEDAPs	272
9.2	SEDAPs Execution Semantics	276
9.3	Correspondence Between SEDAPs and DCDSs	278
9.4	Verifying SEDAPs	279
9.5	From Theory to Practice: SEDAPs Instantiation	281
10	CONCLUSION	301
10.1	Summary	301
10.2	Discussion and Related Works	302
10.3	Future Works	304
	BIBLIOGRAPHY	309

INTRODUCTION

Data and processes are golden ingredients for any information system. As usual, data are simply facts that might be used for a specific purpose, while a (business) process is a sequence of actions/activities that are performed in order to achieve a certain (business) goal, and that might also manipulate data during its execution. Within an information system, data are also considered as the elements that characterize the *static* aspect of the system, while processes characterize the *dynamic* aspect of the system. Due to the importance of data, they are even often considered as the driver of an organization. In fact, typically many prominent and critical (business-related) decisions within an organization are made based on the data. On the other hand, processes are also vital for any competitive business. They differentiate between good and outstanding business performance. Hence, it is inevitable that data and processes are notable aspects within information systems that influence the performance of organizations.

Although data and processes are fundamentally two different entities, they are tightly connected. However, traditional system modeling approaches model data and processes separately. When it comes to process modeling, people often abstract away the data, and when modeling the data, people often think about the processes only afterwards [156, 154, 94]. This situation might be unsatisfactory. As witnessed by [154, 94, 156, 143, 125, 85], there is evidence of the need to treat both data and processes as first class citizens when building a system. They may even be considered as “*two sides of the same coin*” [154]. Thus, focusing on data and processes separately while designing the system might be insufficient. In fact, considering both data and processes together while designing the system could promote us into a better unified holistic view of the system. Furthermore, it could help us in avoiding various problems of the traditional system modeling approaches that consider these two aspects independently (e.g., the system is inadequately covering some process scenarios [154]).

Along with the need of focusing on both data and processes simultaneously, the *artifact-centric business process* paradigm [147, 125, 85] emerges as a promising approach that combines both static and dynamic aspects while designing a system. It provides a rich and robust model for devising business processes in which data and processes are first class citizens. This initiative was initially pioneered at IBM research¹ [147]. Since then, extensive studies have been accomplished in this area and numerous fruitful outcomes have been achieved (e.g., [37, 5, 30, 31, 79, 108, 107]). Moreover, the artifact-centric paradigm has been successfully applied in various settings (cf. [36, 38, 81]). This line of research is often also called *data-aware (business) processes*.

Orthogonal to processes and data, *ontologies* allow us to have a formal conceptualization of the structural/intensional knowledge about the domain of interest. In particular, what do we mean by knowledge is the universal statements about

¹ International Business Machine (IBM) Corp. - <https://www.ibm.com/>

data. Such statements describe the structure of the domain as well as enable us to infer/derive some implicit information from the explicit one. Typically, ontologies are formalized in logic-based languages (e.g., First Order Logic (FOL), or Description Logic (DL)). As an example, consider a *customer order processing* scenario within a company. In FOL-based ontologies, we can encode domain knowledge saying that “*each assembled order is an order*” as a first order sentence/axiom as follows: $\forall x. \text{AssembledOrder}(x) \rightarrow \text{Order}(x)$. Besides enabling us to conveniently structure the domain knowledge, a crucial advantage of ontologies is that they allow us to reason about the domain. For instance, in our example, whenever we know that something is an assembled order, we can infer that it is also an order. Since fundamentally ontology captures the structural knowledge of the domain of interest, we often also consider it as the structural knowledge component of a system.

Looking at ontologies and the artifact-centric approach, there are some researches on data-aware processes formalisms that take into account ontologies (e.g., [22, 121, 20]). Besides allowing us to focus simultaneously on data and processes, the proposed framework enables us to incorporate the domain knowledge inside the designed system and leads us to a semantically-rich system.

When it comes to the need of ensuring the correctness of the developed system, there are various techniques that are usually applied such as (software/system) testing, peer review, simulation and formal verification. The choice of the method is typically based on the complexity of the system as well as the required degree of safety. Each of those techniques has its own advantages and disadvantages. For instance, testing might be easier to do than formal verification, but is in general less reliable. As stated by the famous computer scientist E. Dijkstra, “*Testing can only show the presence of errors, but not their absence*”. In fact, as reported in the survey of artifact-centric business processes models [125], formal verification for artifact-centric systems is an important research direction aimed at establishing sophisticated techniques to analyze the correctness of data-aware business processes. Model checking [27] is a widely studied and successful formal verification technique, see, e.g., [82] for notable success stories. However, the interactions between data and processes typically makes the problem more difficult since it makes the system in general become infinite states. Thereby typical model checking techniques for finite state systems are inapplicable.

In this thesis, motivated by various works on data, processes and ontologies, we focus on the formal verification of several variants of data-aware business processes that are enriched with ontologies. It is noteworthy to remark that this line of research opens up various fascinating connections among diverse research areas such as Databases, Formal Verification, Model Checking, Business Process Management, Knowledge Representation, and specifically Description Logics, and Reasoning About Actions.

The rest of the chapter is organized as follows: In Section 1.1 we briefly overview some studies related to data, processes and knowledge as well as their blending. We then continue by elaborating our research challenges in Section 1.2, and exhibit the core results within this thesis in Section 1.3. We conclude this chapter by listing the publications of the results from this thesis in Section 1.4 as well as providing a concise outlook to the thesis structure in Section 1.5.

1.1 The Meet Up Between Data, Processes and Knowledge

Over the years, there has been plenty of effort in providing means to model the structure of data. This brought us a plethora of data modeling languages such as UML [177, 104], ER diagrams [97], and ORM [118, 119]. Several tools also have been developed in order to ease data modeling (e.g., Rational Rose, Enterprise Architect). Moreover, various researches have been conducted within this area such as *(i)* performing a comparative analysis among different modeling languages [120], *(ii)* studying the correspondence between a certain logic and a particular data modeling language [105], *(iii)* establishing an automated reasoning technique to reason about a specific modeling language [34], etc.

On the other hand, numerous works are concerned with the problem of establishing mechanisms to specify (business) processes. Various approaches for modeling processes have been studied/proposed such as Petri Nets, BPMN, Workflow Pattern, YAWL, and BPEL, (cf. [148, 179, 172]). Some studies on critically comparing or surveying various approaches for business processes modeling can be found in [41, 109, 137, 153].

Ontologies have become a substantial research direction within the area of knowledge representation, which is traditionally concerned with the problem of representing possibly complex knowledge about a domain of interest. Various approaches have been proposed, such as semantic networks [184, 130, 168], frame based systems [101], and logic based approaches [12, 16]. In computer science, knowledge representation focuses on how to represent knowledge such that it is effectively and efficiently machine processable. This leads to the important task of reasoning over the known facts in order to infer unknown/implicit facts from the existing knowledge. Various languages for expressing ontologies have been proposed, notably Description Logics (DLs) [12], and Datalog [44, 45]. The researches on ontologies typically deal with the trade off between expressivity and the computational complexity of inference. Trivially, more expressive languages make reasoning more difficult and vice versa.

In the remainder of this section, we briefly overview various researches that take into account the combination among data, processes or knowledge.

1.1.1 The Marriage Between Processes and Knowledge

A combination between processes and knowledge has been carried out in the context of semantic web services [141, 171]. The idea of semantic web services is to provide a semantic markup on web services to make them understandable and processable by machines. These semantic markups give more information about the services in a machine processable format. One of the advantages from this proposal is enabling automated web service discovery, execution, composition and interoperation. In this context, the web services are the processes and the semantic markups are the knowledge, and the area can be considered as a proposal to get benefits from their combination. One interesting line of research related to semantic web service is that of composition (cf. [129, 142]), which is concerned with the problem of how to compose available services in order to realize a requested, but still unavailable, service.

Another line of research that marries processes and knowledge is that of Semantic Business Processes (SBP), whose basic idea is to adopt semantic technologies for

Business Process Management (BPM) [180, 178]. The motivation comes from the need and the lack of machines accessible semantics in the current business process representation [123]. Adopting semantic technologies might help the automation of many tasks related to BPM [181]. This leads to a research area called Semantic Business Process Management (SBPM) (see [123, 181]). In [123], the authors argue that the lack of machine-readable semantics in current business process representation is the major obstacle in the automation of business process management and they point out that the semantic web and semantic web service technology might provide the necessary tools. Hence, they propose to combine the techniques in SWS and BPM. Continuing the vision of SBPM in [123], the work in [181] describes how ontologies and semantic web service technologies can be used in the BPM lifecycle (process modeling, implementation, execution, and analysis). It also identifies functional requirements of SBPM as well as their benefits, which are mostly about the support of process automation. Some other work on SBPM can be found in [92] which is about mining and monitoring of the process, which are important parts of analysis in the BPM lifecycle. We mention also work about measuring the similarity between SBP [95] and a proposal on a framework for compliance management of SBP [127].

1.1.2 The Marriage Between Data and Knowledge

An extensive study on the marriage between data and knowledge is attested by the research on Ontology Based Data Access (OBDA) [139, 124, 53, 151, 159, 161, 49, 152, 169, 165, 40, 46, 47, 69, 117, 128, 157, 158]. The idea is to provide a conceptual view over (existing) data repositories through ontologies that abstract away from how such data are maintained. Technically speaking, such approach adds an ontology over the data repositories, which captures the domain of interest, and then we can query the data repositories through the ontology. Within this setting, we obtain:

- More sophisticated access to data repositories.
- Sophisticated query answering ability, which enables us to deal with incomplete information and to infer some facts that are not stated explicitly in the data.
- An ability to impose constraints on the data over the conceptual level.
- A high level abstraction that hides the low level details on how the data are stored.
- A unified view on multiple data sources through the ontology.

However, these advantages do not come for free, and various efforts have been made to overcome all challenges such as

- finding the right formalism for the ontology,
- dealing with performance,
- tackling the impedance mismatch problem. I.e, the problem that arises because of the mismatch between what is stored in the data repositories (i.e., values) and in the ontologies (i.e., abstract objects). Such situation demands a mechanism to establish the links between the values in the data repositories and the objects in the ontologies.

Not only theoretical results have been achieved, but also intensive efforts have been put in realizing these concepts into implemented systems (cf. [161, 165, 160, 70, 56]).

1.1.3 The Marriage Between Data and Processes

The marriage between data and processes has been exhibited by various studies in the area of *data-aware business processes*. A survey on this area can be found in [62]. Moreover, some results as well as research directions and challenges specific on the artifact-centric approach can be found in [125, 85].

In artifact-centric approach, the key business-relevant entities are modeled as (*business*) *artifacts*, and an artifact itself is constituted by an *information model* and a *lifecycle*. The former captures the artifact’s relevant data while the latter characterizes the evolution of an artifact (i.e., specifies the permitted ways to progress the information model). One can also say that the lifecycle of an artifact captures the possible “business-relevant stages” as well as their possible changes (from one stage to another) during the evolution of an artifact. During their evolution, the data within an artifact are also manipulated. An artifact-centric system is then constituted by a set of artifacts that might interact with each other and evolve over time. As a simple example, consider an artifact CUSTOMERORDER that represents a business entity that captures the order of a customer. The information model of CUSTOMERORDER might contain the data about the corresponding order (e.g., a list of ordered products). On the other hand, the lifecycle of CUSTOMERORDER characterizes how a CUSTOMERORDER might evolve from one stage to another one, such as from the stage of an order being received to the stage of an order being processed.

Many variants of artifact-centric systems have been studied. Some of the early works on this paradigm are presented in [107, 108]. In those works, the authors investigate artifact-based systems where the information model is constituted by a tuple of typed-attributes. To characterize the lifecycle of artifacts, the framework uses finite state machines. These works mainly focus on investigating static analysis techniques for the artifact-centric framework. Still among the early studies on the artifact-centric paradigm, the work in [93] investigates artifact systems that are equipped with a static relational database (i.e., it stays fixed during the system evolution). The evolution of the system is then characterized by the services that manipulate updatable data inside the existing artifacts. This work investigates the decidability boundaries for the verification of First Order LTL formulas over this setting.

The works in [5, 4, 6] consider artifact-centric systems that are constituted by a set of (interacting) XML-based documents called Active XML (AXML) documents [2, 3]. An AXML document is an XML-based document that may contain embedded function, and that evolves over time based on the result of such function calls. The function calls are differentiated into internal and external function calls. The former do local computations while the latter interact with users or other services. The interesting task in AXML-based artifact-centric systems is to analyze their behavior, which is characterized by the evolution of the documents. In particular, they intend to verify temporal properties over the runs of the system. The temporal properties are specified by the temporal logic Tree-LTL, in which the atomic properties are tree-like patterns that can be checked over the state of the system and the temporal parts are as in the usual Linear Temporal Logic (LTL) [27].

The research in [21] studies artifact-centric systems where data are modeled by relational databases [97]. The authors of [21] primarily focus on investigating the

problem of verifying temporal properties over the evolution of the system that are expressed in a first-order variant of μ -calculus [43].

Still within the spirit of the artifact-centric paradigm, the works in [126, 87] propose the so-called Guard-Stage-Milestone (GSM) as a framework for modeling/specifying artifact-centric systems. GSM is equipped with a formal execution semantics [87], which unambiguously characterizes the artifact progression in response to external events. Notably, several key constructs of the emerging OMG standard on Case Management and Model Notation² have been borrowed from GSM.

Another interesting research direction is the research on the artifact-centric model in the context of multi-agent systems (cf. [30, 32, 115]), leading to the so-called Artifact-Centric Multi-Agent Systems (AC-MAS) framework. In this setting, each of the agents has some internal data stored within them. The system evolution is then characterized based on the actions that are performed by agents, which involve both agent interactions and data manipulation. The main reasoning task that was tackled is temporal properties verification over the system. In addition to theoretical results, a model checker for AC-MAS also has been developed (cf. [115]). Notably, the work in [115] extends GSM towards the setting of multi-agent system.

Other prominent results on the marriage between data and processes are provided by [24, 23] who propose a formal framework called Data-Centric Dynamic Systems (DCDSs). DCDSs basically capture the evolution of a system that is characterized by the manipulation of a relational database by processes (actions). The processes in DCDSs are declaratively specified in terms of condition-action rules that tell when and how an action can be executed. A fascinating result on the decidability of verification of first-order variants of μ -calculus properties over DCDSs has been obtained. Some attempts in implementing DCDS have also been carried out (cf. [163, 73, 74]).

1.1.4 When Data, Processes and Knowledge Meet Up

The meet up among data, processes and knowledge (specifically, ontologies) can be observed in the work on semantic artifacts [20] and also in those works that combine Knowledge Bases (KBs) and actions [121, 146, 63, 14, 13, 186].

In [20], the authors propose artifact-centric systems formed by semantic artifacts, which utilize Description Logic (DL) KBs as the mechanism to keep artifact relevant information in a semantically-rich form. As usual, a DL KB is constituted by an ABox that stores the data and a TBox that captures the domain knowledge. The progression mechanism of a semantic artifact system is provided in a declarative manner using condition-action rules similar to [21]. Furthermore, [20] studies the problem of model checking over the semantic artifact systems temporal properties that are specified using a first order variant of μ -calculus [43]. Although in general the problem is undecidable, [20] has identified a syntactic restriction that guarantees decidability of verification based on the notion of weak-acyclicity in data exchange [99].

The works [14, 15] introduce a DL-based action formalism. The semantics of DL-based actions is specified in terms of manipulation of DL interpretations (which are first-order interpretations of the unary and binary predicates corresponding to concepts and roles, respectively). Concerning reasoning, [14, 15] study the projection

² <http://www.omg.org/spec/CMMN/>

problem, i.e., the problem of checking whether the execution result of a certain sequence of actions satisfies a given DL formula (assertion). Building on [14, 15], the works [13, 186] study Golog programs [134] in which the atomic actions are formalized as DL-based actions. Within this setting, [13, 186] tackle the problem of verifying temporal properties over the execution of Golog program with respect to the given DL KB.

In [22, 121], the authors propose a formal framework, named Knowledge and Action Bases (KABs), which allows one to capture the manipulation of a DL Knowledge Base over time. The dynamic aspect of KABs is characterized by condition-action rules that, together with the data manipulated during the system evolution, determine the possible sequences of actions that can be executed over the KB. In contrast to [14, 15, 13, 186], where actions directly manipulate DL-interpretations, the work in [22, 121, 146] adopts the Levesque functional approach [135] in defining the execution semantics of actions. Under this approach, the KB provides two operations, ASK to extract relevant information, and TELL to assert new knowledge, and actions rely on such operations for their execution. Specifically, in KABs, the ASK operation corresponds to the computation of *certain answers* to queries over the KB, and the TELL operation corresponds to asserting the ABox facts that should hold in the resulting state. During action execution, calls to external services might be issued in order to acquire *new values* from outside of the system. As a consequence, the number of possible states is not bounded a priori. Roughly speaking, the calls to external services can be used to model the interaction with external systems/entities as well as user input that might inject new values to the system. The execution semantics of a KAB is provided by a possibly infinite state transition systems in which each transition represents an action execution and each state contains a KB. Regarding reasoning, [22, 121] study the verification of temporal properties over the evolution of KABs, where the temporal properties are specified by a first order variant of μ -calculus. Although in general verification is undecidable, decidability has been obtained under suitable restriction based on the notion of *weak acyclicity* that is borrowed from the work on data exchange [99].

1.2 Research Challenges

Many data-aware processes formalisms that have been studied so far consider a simple formalism for specifying the progression mechanism. For instance, [24, 23, 21, 20] only consider condition-action rules to specify when and how an atomic action can be executed. Although this approach is quite expressive, one might desire a better control in specifying the desired order of actions (e.g., to choose one action or another based on the result of a condition checked over the current state, or to specify that a certain sequence of actions should be executed as long as a specified condition holds). Thus, a more sophisticated formalism is required to specify the system dynamics at a higher-level of abstraction. Example 1.1 illustrates this issue.

Example 1.1. Consider an *order processing scenario* in a furniture provider enterprise. Consider the actions/operations `approveOrder` and `prepareOrder`, where an execution of `approveOrder` approves a single *received order* and an execution of `prepareOrder`

prepares a single *approved order*. Suppose that we want to strictly enforce that each single execution of `approveOrder` must be followed by `prepareOrder`. In the typical condition-action rules formalism, we can only specify the pre-condition of each action/operation. Therefore, in such formalism we have to specify that at the end of the execution of `approveOrder`, it should make the pre-condition of `prepareOrder` become satisfied and, the pre-condition of `approveOrder` should not be satisfied. Considering that there could also be other actions/operations, then we also need to make sure that the pre-conditions of all other actions are not satisfied. Thus, looking at this situation, it might be desirable if we can directly specify such *expected sequence of actions/operations*, i.e., by directly saying

`approveOrder; prepareOrder`

which semantically means that every execution of `approveOrder` must be followed by `prepareOrder`.

As another natural example, consider the situation where we need to specify that a process should be repeated as long as a particular condition holds. For instance, after delivering a processed order, the whole process should go back to the beginning and repeat the whole process until there is no more unprocessed order. It might be desirable if we could specify such situation in a high level manner by a kind of “while loop” construction. E.g.,

```
while "there exists an unprocessed order" do {
    approveOrder; prepareOrders; ...; deliverOrders
}
```

Concerning inconsistency management, the majority of approaches dealing with verification in data-aware processes assume a rather simple treatment. In particular, they simply reject inconsistent system states that are produced by the effects of action executions (see, e.g., [93, 24, 121, 20]). In general, this mechanism is not satisfactory, since the inconsistency may affect just a small portion of the entire data, and thus should be treated in a more careful way. This is in line with what is done in numerous researches that specifically deal with inconsistencies (cf. [131, 39, 35, 59]). Example 1.2 illustrates this issue.

Example 1.2. Continuing Example 1.1, consider that there is a process of designing and assembling an order. Suppose that to enforce segregation of duty, we have a *domain knowledge (constraint)* in our system saying that “*a product designer is not a product assembler, and vice versa*”. Suppose that we have the fact that “*john is a product designer*” (i.e., `Designer(john)`). Now, suppose that after an execution of an action α , we have a new fact that “*john is a product assembler*” (i.e., `Assembler(john)`). Hence, we have that the constraint above is violated (i.e., at this state the system encounters an inconsistency). In this situation, instead of disallowing the execution of α that leads into this inconsistent state (i.e., rejecting this inconsistent state), it

might be desirable to repair the inconsistent state, for instance by throwing the older fact (i.e., `Designer(john)`). In fact, it could be the case that John has been recently change his role into designer, but the system might not yet been updated.

Many works on data-aware processes that incorporate structural domain knowledge typically assume that such knowledge remains fixed along the system evolution (e.g., [63, 146, 121]), i.e., that it is independent from the actual system state. However, this assumption might be too restrictive, since specific knowledge might hold or be applicable only in specific, *context-dependent* circumstances. Ideally, one should be able to form statements that are known to be true in certain cases, but not necessarily in all. Example 1.3 illustrates this issue.

Example 1.3. Continuing Example 1.2, suppose that we want to have such a constraint only hold during the *normal season*, but during the *peak season*, to enforce efficiency, we want to have that “*each product designer is a product assembler*”. Hence, in this situation, it might be desirable to contextualized our domain knowledge such that

1. in the context of *normal season* a designer must not be a product assembler (and vice versa),
2. in the context of *peak season* each designer is also a product assembler

As witnessed by numerous works on data-aware processes (see e.g., [93, 30, 24, 121, 146]), the verification problem in this setting is in general difficult (more precisely, undecidable without suitable restrictions) since the number of systems states is in general infinite. Thus, off the shelf model checking technique for finite state system cannot be used directly. The situation becomes even more challenging when we also need to deal with inconsistencies and/or take into account the presence of contextual information.

In some formalisms of data-aware processes, the information model typically relies on relatively simple structures, such as tuples of typed-attributes (e.g. [107, 108, 126, 87]). This situation might cause an abstraction gap between the high-level conceptual view that business stakeholders have, and the low-level representation of information. In addition, the data layer within the system might be complicated and difficult to interact with. In this light, there is a need to have a high level conceptual view over the system evolution.

In this thesis, we aim at addressing all the issues mentioned above, by proposing novel extensions of existing models for data-aware business processes, and by studying how these extensions affect the problem of formal verification of expressive temporal properties. In the remaining part of the chapter, we discuss in detail the original contributions that we have provided along these lines.

1.3 Contributions

As a first broad contribution of this thesis, we introduce and study several variants and extensions of the formalism of KABs. Specifically, the extensions we introduce are the following:

1. A formal framework, namely Golog-KABs (GKABs), for specifying *semantically-rich data-aware business processes* that is obtained by leveraging on the current state of the art data-aware processes equipped with ontologies.
2. Several variants of *inconsistency-aware Golog-KAB*, which extend GKABs by incorporating various inconsistency handling mechanisms that had been proposed in the literatures.
3. An extended version of GKABs, namely *Context-Sensitive Golog-KABs* (CSGKABs), which takes into account contextual information during the evolution.
4. Several variants of *inconsistency-aware context-sensitive Golog-KAB*, which are obtained from CSGKABs by incorporating various inconsistency management mechanisms.
5. An extension of GKABs, called *Alternating GKABs*, that separates the sources of non-determinism within a single step of evolution and allows for a more fine-grained analysis on the system evolution, while also employing sophisticated inconsistency handling mechanism and taking into account contextual information.
6. A novel framework, called *Semantically-Enhanced Data-Aware Processes* (SEDAPs), which enables us to have a high-level conceptual view over the evolution of a data-aware business processes by utilizing ontologies.

We observe that this thesis establishes two different approaches in devising a semantically-rich data-aware business processes. One, based on GKABs and their variants, in which we have a KB that evolves under the effect of actions, requires us to specify the system from scratch. The other one, namely SEDAPs, enables us to enhance existing data-aware processes systems towards a semantically-rich system by connecting an ontology via mappings to a traditional relational data layer that evolves under action execution.

Within all of the settings above, we tackle the problem of verification of temporal properties over the system executions. This task is more challenging than in the basic setting of KABs, on which we build, since we need to deal with inconsistency in a more sophisticated manner and consider the contextual information. In the following sub-sections, we provide more details on each of these contributions.

1.3.1 Golog-KABs (GKABs)

Here we devise a formal framework for specifying *semantically-rich data-aware business processes systems* by leveraging on the current state of the art data-aware pro-

processes system equipped with ontologies [121, 22, 146, 63]. Specifically, we build on the Knowledge and Action Bases (KABs) framework that was initially proposed in [121, 22]. Fundamentally, KABs provide a semantically rich representation of a domain in the form of a KB expressed in the lightweight DL *DL-Lite* [50], while also simultaneously taking into account the dynamic aspects of the modeled system. As usual, the *DL-Lite* KB is constituted by a *TBox* that captures the intensional knowledge about the domain and an *ABox* that keeps the data (extensional parts). The execution semantics of a KAB is given in terms of a (possibly infinite) transition system, in which each state is labeled by a DL KB and each transition represents the manipulation of the ABox by an action. Concerning action specification, rather than following the original KABs [121, 22], in which at each action execution the state is reconstructed from scratch, we adopt the action formalism in [146], in which one specifies only the facts to add and those to delete from the current state. Similar to KABs, an action execution might issue external service calls that might inject fresh values (constants) into the system. Roughly speaking, the calls to external services can be used to model the interaction with external systems/entities as well as user input. As for the execution semantics w.r.t. service calls, instead of following [121, 22], we use the service call evaluation semantics as in [24, 23], which is considered to be less abstract, more natural, and closer to reality. I.e., we evaluate the service calls in the sense that we substitute each service call with a concrete value when constructing the transition system. Since we use KABs that are slightly different from their original version in [121, 22], in Chapter 3 we show that the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ properties over KABs can be reduced to the corresponding verification of $\mu\mathcal{L}_A$ over DCDSs [24], where $\mu\mathcal{L}_A^{\text{EQL}}$ and $\mu\mathcal{L}_A$ are variants of first order μ -calculus [43] (one of the most powerful temporal logics, which subsumes LTL, PSL, and CTL* [83]). The difference between $\mu\mathcal{L}_A$ and $\mu\mathcal{L}_A^{\text{EQL}}$ formulas is in the atomic parts of the formulas. The former consider Domain Independent First Order Logics queries [1] as the atomic components of the formulas while the latter consider Domain Independent EQL-Lite (UCQ) [51] queries. The reduction also preserves *run-boundedness*, which is a restriction that guarantees the decidability of DCDSs verification. Thus, exploiting the results on verification of run-bounded DCDSs, it follows that the verification of run-bounded KABs is decidable and can be reduced to standard finite state model checking.

In this thesis, we enrich KABs with a high-level, compact action language inspired by a well-known action programming language in the area of Artificial Intelligence (AI), namely Golog [134]. We call the resulting formalism *Golog-KABs* (GKABs). Thus, instead of using simple condition-action rules as in KABs, the progression mechanism in GKABs is specified using Golog programs. This allows modelers to conveniently specify the processes at a high-level of abstraction and represent the dynamic aspects of the systems much more compactly (cf. Example 1.4). Roughly speaking, the Golog program characterizes the evolution of a GKAB by determining the possible orders of action executions that evolve the KB over time.

Example 1.4. Recall our Example 1.1, using Golog constructs, we can specify such a sequence of actions as well as the repetition of operations in a convenient way.

Using the Golog construct “;” we can specify the sequence of actions `approveOrder` and `prepareOrder`, i.e.,

`approveOrder; prepareOrder`

Furthermore, we can also specify the repetition of operations using the construct

while φ **do** δ

which we can use to express that δ will be executed as long as φ satisfied. See Chapter 4 for more details.

To elegantly accommodate various ways of updating the ABox, we introduce a parametric execution semantics of GKABs. Technically, we adopt Levesque’s functional approach, i.e., we assume that a GKAB provides two operations:

- ASK, to answer queries over the current KB;
- TELL, to update the KB (ABox) through an atomic action.

In this work, the ASK operator corresponds to the *certain answers* computation. The TELL operation is parameterized by *filter relations*, which are used to refine the way in which an ABox is updated, based on a set of facts to be added and deleted (that are specified by the action).

In this light, filter relations provide an abstract mechanism to accommodate in the execution semantics several inconsistency management approaches based on the well-known notion of repair [132, 133, 54]. Basically, we can obtain various execution semantics for GKABs, including *inconsistency-aware semantics*, by simply defining different kinds of filter relation. For instance, we define *GKABs with standard execution semantics*, briefly *S-GKABs*, by defining a filter relation f_S that updates an ABox based on the facts to be added and deleted, and does nothing w.r.t. inconsistency (i.e., updates that lead to an inconsistent state are simply rejected).

Concerning the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ properties over S-GKABs, we have shown that we can reduce this problem to verification of KABs and vice versa. To encode KABs into S-GKABs, we simulate the standard execution semantics using a Golog program that runs forever to non-deterministically pick an executable action with parameters, or stops if no action is executable. For the opposite direction, the key idea is to inductively interpret a Golog program as a structure consisting of nested processes, suitably composed through the Golog operators. We mark the starting and ending point of each Golog subprogram, and use accessory facts in the ABox to track states corresponding to subprograms. Each subprogram is then inductively translated into a set of actions and condition-action rules, encoding its entrance and termination conditions.

1.3.2 Inconsistency-Aware GKABs

We introduce GKABs with inconsistency-aware semantics by exploiting the filter relations (i.e., we introduce various kind of filter relations and plug them in into GKABs). By incorporating inconsistency-aware semantics, we allow each action that leads to an inconsistent state and then we repair the inconsistency. Figure 1 gives an illustration

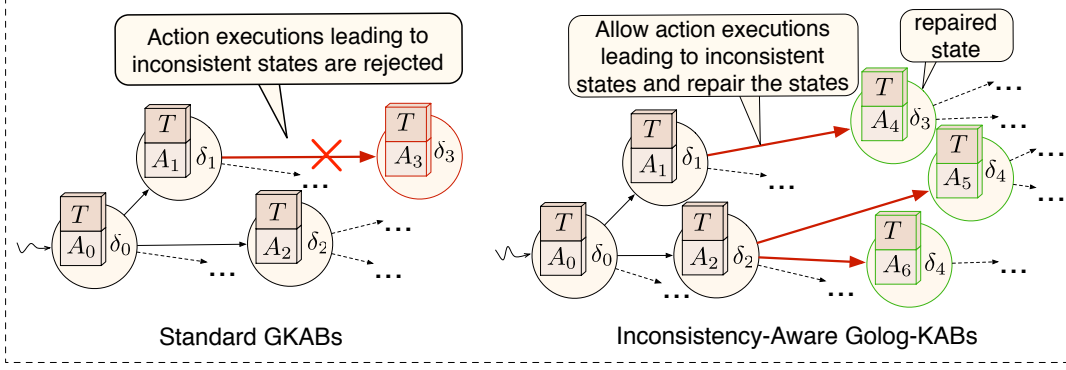


Figure 1: An illustration of Inconsistency-Aware GKABs execution compare to Standard GKABs execution (Note: T is a TBox, A_i is an ABox, and δ_i is the remaining program to be executed).

of this setting. Technically, we introduce filter relations B-filter f_B , C-filter f_C , and B-evol filter f_E , where

- f_B incorporates the ABox Repair (AR) semantics in [132]. Here we call such approach *bold-repair* (*b-repair*), where a b-repair of an ABox A w.r.t. TBox T is a maximal (w.r.t. set containment) subset of A that is consistent with T .
- f_C incorporates the Intersection ABox Repair (IAR) semantics in [132]. Here we call such approach *certain-repair* (*c-repair*), where a c-repair of an ABox A is an ABox that is obtained by intersecting all b-repairs of A w.r.t. T .
- f_E updates the ABox using the bold semantics of KB evolution [54]. In this approach, if an inconsistency arises due to an update, newly introduced assertions are preferred to those already present in the current ABox.

We call the GKABs adopting the execution semantics obtained by employing those filter relations *B-GKABs*, *C-GKABs*, and *E-GKABs*, respectively. We group them under the umbrella of *inconsistency-aware GKABs* (*I-GKABs*). Example 1.5 provide a high level illustration of this setting.

Example 1.5. Recall Example 1.2 where we have an inconsistent state containing the facts *Designer(john)* and *Assembler(john)*. Suppose that these two facts are the only facts that cause inconsistency, then

- In B-GKABs (i.e., GKABs that employ AR semantics for updating the ABox), we repair that state and produce two repair states. One repair state containing the fact *Designer(john)* while the other one containing the fact *Assembler(john)*. I.e., it explores all possible repairs.
- In C-GKABs (i.e., GKABs that employ IAR semantics for updating the ABox), we repair that state and produce a repair state containing neither *Designer(john)* nor *Assembler(john)*. I.e., it only keeps the facts that do not involve in inconsistency.
- In E-GKABs (i.e., GKABs that employ bold semantics of KB evolution for updating the ABox), we repair that state and produce a repair state containing

Assembler(john) (i.e., throw away **Designer(john)**). It reflects the situation where new facts are considered to be more correct.

With respect to verification of $\mu\mathcal{L}_A^{\text{EQL}}$ properties over the various types of GKABs introduced so far, we have proved the results summarized in Figure 2, where an arrow indicates that we can reduce verification in (G)KABs in the source to verification in (G)KABs in the target. Furthermore, the semantic property of *run-boundedness*

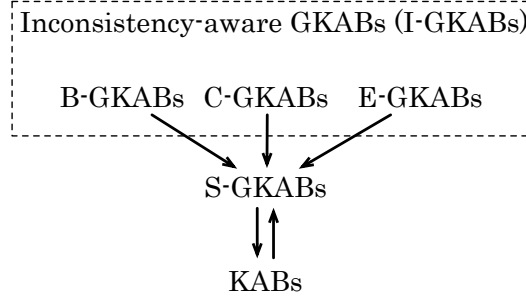


Figure 2: Reductions from I-GKABs (i.e., B-GKABs, C-GKABs, and E-GKABs) to KABs

(which guarantees the decidability of KAB verification) [24] is preserved by all our reductions. Thus, it follows that verification of $\mu\mathcal{L}_A^{\text{EQL}}$ properties over run-bounded S-GKABs and I-GKABs is decidable, and reducible to standard μ -calculus finite-state model checking. For all reductions from I-GKABs to S-GKABs, our general strategy is to show that S-GKABs are sufficiently expressive to incorporate the repair-based approaches, so that an action executed under certain inconsistency-aware semantics can be compiled into a Golog program that applies the action with the standard semantics, and then explicitly handles the inconsistency, if needed.

1.3.3 Context-Sensitive GKABs

As the next contributions, we extend GKABs towards Context-Sensitive GKABs (CSGKABs), which allow us to incorporate contextual information within the system. The context might change during the system evolution and influences the system execution in several ways such as:

- determining relevant TBox assertions at each state (i.e., TBox changes along the system execution depending on the context), and
- influencing the decision about action executability.

As a consequence of the TBox changes, essentially context also indirectly affects the results of query answering over the KB. Example 1.6 provide a high level intuition of CSGKABs, and Figure 3 illustrates the execution of CSGKABs.

Example 1.6. Recall Example 1.3, in CSGKABs, we can capture the situation of *normal season* and *peak season* as “context”. Moreover, we can encode those context-dependent knowledge (i.e., “*each product designer is a product assembler*” and “*a*

product designer is not a product assembler (and vice versa)”) such that they only hold on the corresponding desired context. Please see Chapter 6 for more details.

Concerning execution semantics, it is worth mentioning that we lift GKABs into CSGKABs by also retaining their parametric execution semantics. Therefore, we can easily define various ways of updating the ABox in CSGKABs by simply “shaping” the filter relation, which is a great basis for integrating various inconsistency management mechanisms into CSGKABs.

Regarding verification, to specify the properties to be verified, we consider a context-sensitive temporal logic $\mu\mathcal{L}_{\text{CTX}}$, which extends $\mu\mathcal{L}_A^{\text{EQL}}$ with the possibility of having also “context expressions” as an atomic part of the formula. It follows that, using $\mu\mathcal{L}_{\text{CTX}}$ we can also say something about contextual information inside the properties that we want to verify. In Chapter 6, we study the verification of CSGKABs with standard execution semantics, briefly *S-CSGKABs*, that are obtained by using the standard filter relation. To cope with the problem of verifying $\mu\mathcal{L}_{\text{CTX}}$ over S-CSGKABs, we reduce the problem to the corresponding $\mu\mathcal{L}_A^{\text{EQL}}$ verification problem over S-GKABs.

1.3.4 Inconsistency-Aware Context-Sensitive GKABs

We also study the combination of CSGKABs and various inconsistency management mechanisms (as in I-GKABs), which led us to the formalization of Inconsistency-aware Context-sensitive GKABs. In particular, similar to the way of obtaining I-GKABs, we employ three filter relations that incorporate the b-repair, c-repair, and bold-evolution computations. We call CSGKABs adopting the execution semantics obtained by injecting those filter relations B-CSGKABs, C-CSGKABs, and E-CSGKABs, respectively. We group them under the umbrella of Inconsistency-aware Context-sensitive GKABs (I-CSGKABs).

For the verification of $\mu\mathcal{L}_{\text{CTX}}$ over I-CSGKABs, we show that the verification of $\mu\mathcal{L}_{\text{CTX}}$ over B-CSGKABs, C-CSGKABs, and E-CSGKABs can be reduced to the corresponding verification of $\mu\mathcal{L}_A^{\text{EQL}}$ over S-GKABs. Furthermore, all our reduc-

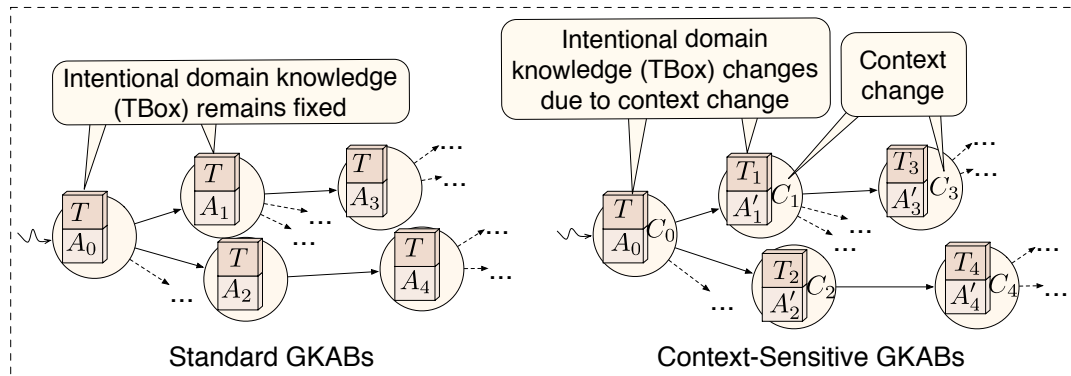


Figure 3: An illustration of Context-Sensitive GKABs execution compare to Standard GKABs execution (Note: T is a TBox, A_i is an ABox, and C_i is a context).

tions also preserve run-boundedness. It follows that the verification of run-bounded S-CSGKABs, B-CSGKABs, C-CSGKABs, and E-CSGKABs are decidable and reducible to the standard finite state model checking.

1.3.5 Alternating GKABs

As a deeper study on GKABs, we introduce AGKABs, which separate sources of non-determinism during the computation of successor states. Those sources of non-determinism are:

1. the choice of grounded actions,
2. the choice of service call results,
3. the choice among all possible new contexts, and
4. the choice of repaired ABoxes when there are several possible repairs (which is the case for b-repairs).

In I-CSGKABs, we encapsulate the computation of all of those sources of non-determinism in a single transition (i.e., roughly speaking, in a single transition, non-determinism can be caused by those four sources). In AGKABs, we separate them such that each state only has one possible source of non-determinism (one of those four sources). Figure 4 gives an illustration of AGKABs execution, and Example 1.7 provides an example that gives an intuition on those sources of non-determinism.

Example 1.7. Recall our Examples 1.1 to 1.6.

- Concerning non-determinism from the choice of grounded actions, now let the action `approveOrder` has a parameter that is the order to be approved (i.e., we have `approveOrder(x)` where x is the order to be approved). Suppose that there are three received orders (let say chair, table, and cupboard), then we can execute the action `approveOrder/1` with several possible arguments (i.e., either `approveOrder(chair)`, `approveOrder(table)`, or `approveOrder(cupboard)`). Hence, from that state, we have several choice of grounded actions that we can execute, and it causes non-determinism in the transition system.
- About non-determinism from the choice of service call results, as it has been mentioned above, within an action execution we might issue a service call. Considering the semantics of service calls, since we consider all possible return values of a service call, then it is easy to see that there are many possibilities of service call results and it causes non-determinism in the transition system.
- Regarding the context change, our model also allow non-deterministic changes.
- As for repairs, we could see in Example 1.5 that we could have several possible repairs when we adopt AR semantics.

Thanks to the separation of the sources of non-determinism, we are capable to do a more fine-grained analysis over the system evolution. In particular, we can verify temporal properties that quantify over each source of non-determinism. For instance, we can check a property like “*no matter which action is executed, there exists a service call result in which no matter how the context is changing, there exists a repair that leads us into a certain state that satisfy a certain property*”.

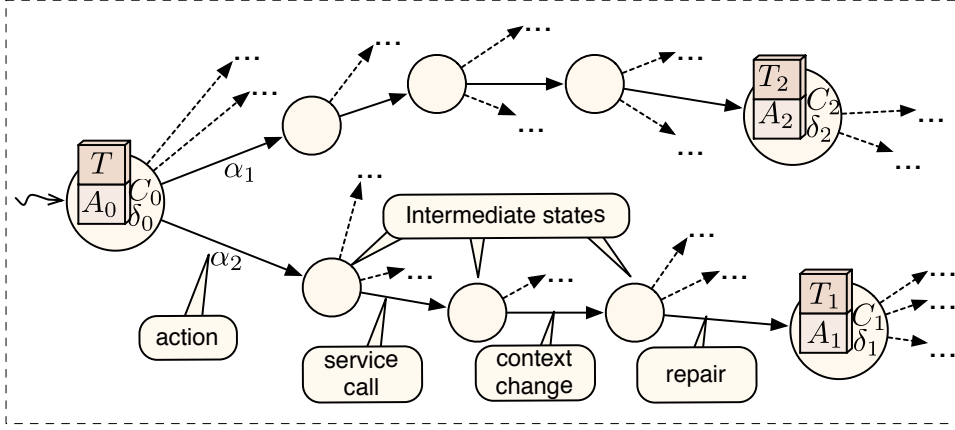


Figure 4: Illustration of Alternating GKABs execution (Note: T is a TBox, A_i is an ABox, C_i is a context, and δ_i is the remaining program to be executed).

Concerning verification, we introduce $\mu\mathcal{L}_{\text{CTX}}^{\text{Alt}}$, which is a fragment of $\mu\mathcal{L}_{\text{CTX}}$ where we always use the modal operators in groups of 4 (e.g., $\langle \rightarrow \rangle [\rightarrow] [\rightarrow] \langle \rightarrow \rangle \Phi$) in order to quantify separately over each source of non-determinism. Similar to I-CSGKABs, we employ three filter relations that incorporate the b-repair, c-repair, and bold-evolution computations, obtaining respectively B-AGKABs, C-AGKABs, and E-AGKABs.

To tackle the problem of $\mu\mathcal{L}_{\text{CTX}}^{\text{Alt}}$ verification over B-AGKABs, C-AGKABs, and E-AGKABs, we prove again that those problems are reducible to the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ over S-GKABs. Also in this case, our reductions preserve run-boundedness, allowing us again to reduce verification to standard finite state model checking.

1.3.6 Semantically-Enhanced Data-Aware Processes

As a further contribution, we devise a novel framework that enables us to enhance the existing data-aware business processes system into a semantically-rich data-aware processes system. In particular we propose *Semantically-Enhanced Data-Aware Processes* (SEDAPs) which are inspired by the research on Ontology-Based Data Access (OBDA) [53], where an ontology is used to provide a conceptual view over (existing) data repositories, to which the ontology is connected by means of mappings. Roughly speaking, SEDAPs can be considered as an extension of DCDSs [24] where the data layer is constituted by an OBDA system instead of simply a relational database. Through the presence of the ontology, a SEDAP provides a unified, high-level conceptual view of the system, reflecting the relevant concepts and relations of the domain of interest and abstracting away how processes and data are concretely realized and stored at the implementation level. This, in turn, is the basis for different important reasoning tasks such as verification of conceptual temporal properties, regulating how new processes can be injected into the system, synthesizing new processes starting from high level conceptual requirements, and reasoning under implicit and incomplete information.

Basically a SEDAP is constituted by three components: (i) an *OBDA system*, which keeps all the data of interest and provides a conceptual view over it in terms of a *DL-Lite_A* TBox; (ii) a *process component* as in DCDSs, which characterizes the evolution

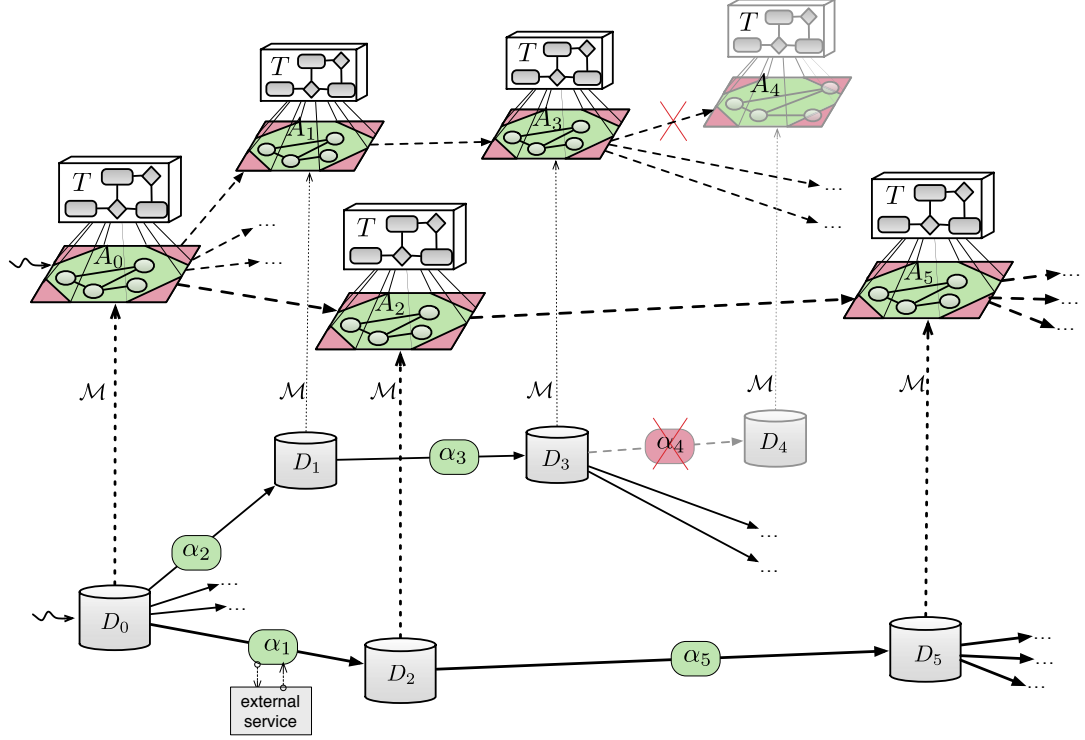


Figure 5: Intuition of SEDAPs setting

(dynamic aspect) of the system; and (iii) an *initial database instance*. Conceptually, a SEDAP separates the system into two layers, the *relational layer* and the *semantic layer*. The relational layer captures the database evolution (manipulation) done by the process execution, while the semantic layer exploits the ontology for providing a conceptual view of the system evolution. This enables us to (i) *understand* the evolving system through the semantic layer, and (ii) *govern* the evolution of the system at the semantic layer by rejecting those process actions that, currently executed at the relational layer, would lead to new system states that violate some constraint of the ontology. Formally, the semantics of SEDAPs is defined in terms of two transition systems: a *Relational Layer Transition System* (RTS) and a *Semantic Layer Transition System* (STS). The RTS is the same as the transition system of a classical DCDS, which captures the evolution of the system at the relational layer, tracking how the database is evolved by the process component. On the other hand, the STS is a “virtualization” of the RTS in the semantic layer and provides a conceptual view of the system evolution. In particular, the STS maintains the structure of the RTS unaltered, reflecting that the process component is executed over the relational layer, but it associates to each state the set of concept and role assertions obtained from the application of the mappings starting from the corresponding database instance. The intuition of the SEDAP setting is depicted in Figure 5.

Within SEDAP, we address the problem of verifying *conceptual temporal properties* that are specified at the semantic layer. Roughly speaking, to tackle the verification problem, we bring down the conceptual temporal property from the semantic layer into the relational layer, by adopting the concept of “rewriting” and “unfolding” in OBDA, and then exploit the decidability results of temporal property verification in

DCDS. I.e., we show that the verification of SEDAPs can be reduced to the verification of DCDSs.

Going beyond theoretical results only, we have instantiated the concept of SEDAPs into a working tool called OBGSM, in which we use the standard Guard-Stage-Milestone (GSM) model [126, 87] to represent the system in the relational layer. OBGSM provides a functionality to translate the temporal property specified at the semantic layer into the temporal property over the relational layer by applying the “rewriting” and “unfolding” technique. It exploits two already existing tools to provide its functionalities:

1. -ONTOP³, a JAVA-based framework for OBDA, and
2. the *GSMC model checker*, developed within the EU FP7 Project ACSI⁴, to verify GSM-based artifact-centric systems against temporal/dynamic properties [31].

OBGSM also becomes a part of EU FP7 Project ACSI deliverable (see [60]), and additionally, we also show how OBGSM can be used in one of the practical use cases of the EU FP7 Project ACSI.

1.3.7 Summary of All Reductions

In addition to all reductions above, we also show that the verification of S-GKABs can be reduced to the corresponding verification of B-GKAB, C-GKAB, E-GKAB, S-CSGKABs, B-CSGKABs, C-CSGKABs, E-CSGKABs, B-AGKABs, C-AGKABs, and E-AGKABs. Thus, summing it up, we have enriched the state of the art data-aware business processes equipped with ontologies so that they can accommodate various prominent scenarios without adding additional computational complexity.

All our reductions are visually summarized in Figure 6, where an arrow indicates that we can reduce verification in the formalism at the source of the arrow to verification in the formalism at the destination of the arrow.

1.4 List of Publications

Some of the core results in this thesis have been published as detailed below:

- In conference proceeding:
 1. Diego Calvanese, Marco Montali and Ario Santoso. *Verification of generalized inconsistency-aware knowledge and action bases*. In Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI), pages 2847-2853, AAAI Press, 2015.
 2. Diego Calvanese and İsmail İlkan Ceylan and Marco Montali and Ario Santoso. *Verification of context-sensitive knowledge and action bases*. In Proceedings of the 14th European Conference on Logics in Artificial Intelligence (JELIA), volume 8761 of Lecture Notes in Computer Science, pages 514-528. Springer, 2014.
 3. Babak Bagheri Hariri, Diego Calvanese, Marco Montali, Ario Santoso, and Dmitriy Solomakhin. *Verification of semantically-enhanced artifact sys-*

³ <http://ontop.inf.unibz.it/>

⁴ “Artifact-Centric Service Interoperation”, see <http://www.acsi-project.eu/>

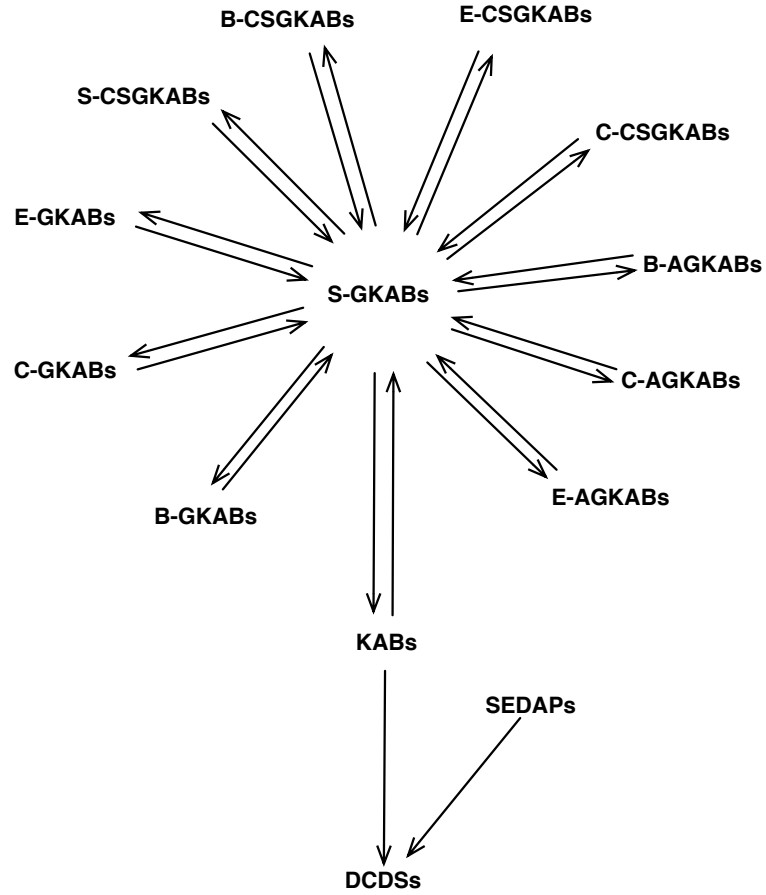


Figure 6: Summary of the reductions developed in this thesis. The meaning of an arrow from formalisms A to formalism B is that verification of A is reducible to verification of B

- tems*. In Proceedings of the 11th International Conference on Service Oriented Computing (ICSOC), volume 8274 of Lecture Notes in Computer Science, pages 600-607. Springer, 2013.
4. Diego Calvanese, Evgeny Kharlamov, Marco Montali, Ario Santoso, and Dmitriy Zheleznyakov. *Verification of inconsistency-aware knowledge and action bases*. In Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI), pages 810-816. AAAI Press, 2013.
 5. Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Marco Montali, and Ario Santoso. *Ontology-based governance of data-aware processes*. In Proceedings of the 6th International Conference on Web Reasoning and Rule Systems (RR), volume 7497 of Lecture Notes in Computer Science, pages 25-41. Springer, 2012.
 6. Ario Santoso. *When data, knowledge and processes meet together*. In Proceedings of the 6th International Conference on Web Reasoning and Rule Systems (RR), volume 7497 of Lecture Notes in Computer Science, pages 291-296. Springer, 2012.
- In workshop proceeding:
 1. Diego Calvanese, Marco Montali and Ario Santoso. *Inconsistency management in generalized knowledge and action bases*. In Proceedings of the 28th International Workshop on Description Logic (DL), volume 1350, 2015.
 2. Diego Calvanese, İsmail İlkan Ceylan, Marco Montali and Ario Santoso. *Adding context to knowledge and action bases*. In Workshop Notes of the 6th International Workshop on Acquisition, Representation and Reasoning about Context with Logic (ARCOE-Logic 2014), volume arXiv:1412.7965 of CoRR Technical Reports, pages 25-36. arXiv.org e-Print archive, 2014. Available at <http://arxiv.org/abs/1412.7965>.
 3. Diego Calvanese, Evgeny Kharlamov, Marco Montali, Ario Santoso, and Dmitriy Zheleznyakov. *Verification of inconsistency-aware knowledge and action bases*. In Proceedings of the 26th International Workshop on Description Logics (DL), volume 1014 of CEUR Electronic Workshop Proceedings, <http://ceur-ws.org/>, pages 107-119, 2013.
 4. Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Marco Montali, and Ario Santoso. *Semantically-governed data-aware processes*. In Proceedings of the 1st International Workshop on Knowledge-intensive Business Processes (KiBP), volume 861 of CEUR Electronic Workshop Proceedings, <http://ceur-ws.org/>, pages 21-32, 2012.
 - Technical Reports:
 1. Diego Calvanese, Marco Montali and Ario Santoso. *Verification of generalized inconsistency-aware knowledge and action bases (extended version)*. CoRR Technical Report arXiv:1504.08108, arXiv.org e-Print archive, 2015. Available at <http://arxiv.org/abs/1504.08108>.
 2. Babak Bagheri Hariri, Diego Calvanese, Marco Montali, Ario Santoso, and Dmitry Solomakhin. *Verification of semantically-enhanced artifact systems*

- (*extended version*). CoRR Technical Report abs/1308.6292, arXiv.org e-Print archive, 2013. Available at <http://arxiv.org/abs/1308.6292>.
3. Diego Calvanese, Babak Bagheri Hariri, Riccardo De Masellis, Domenico Lembo, Marco Montali, Ario Santoso, Dmitry Solomakhin, and Sergio Tesaris. *Techniques and tools for KAB, to manage action linkage with the Artifact Layer - Iteration 2*. Deliverable ACSI-D2.4.2, ACSI Consortium, May 2013.
 4. Diego Calvanese, Evgeny Kharlamov, Marco Montali, Ario Santoso, and Dmitriy Zheleznyakov. *Verification of inconsistency-aware knowledge and action bases (extended version)*. CoRR Technical Report arXiv:1304.6442, arXiv.org e-Print archive, 2013. Available at <http://arxiv.org/abs/1304.6442>.
 5. Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Marco Montali, Marco Ruzzi and Ario Santoso. *Techniques and Tools for KAB, to Manage Data Linkage with the Artifact Layer*. Deliverable ACSI-D2.3, ACSI Consortium, May 2012.

1.5 Organization of the Thesis

We close this chapter by elaborating on the structure of the thesis:

1. In Chapter 1 provides, we provide a high level introduction to the research area, outline the research challenges, and summarize the contributions of this thesis.
2. In Chapter 2, we introduce some preliminaries that are necessary for the comprehension of the thesis. They include some relevant basic notions, agreements on notations, and a quick introduction to (i) relational databases, (ii) *DL-Lite* knowledge bases, (iii) query answering, (iv) history preserving μ -calculus, and (v) Data Centric Dynamic Systems (DCDSs).
3. In Chapter 3, we review the basic notion of Knowledge and Action Bases (KAB) as proposed by [22, 121], and we show how to reduce verification of KABs into verification of DCDSs [24], which, in the end, opens the door for us to use the established results in [24].
4. In Chapter 4, we exhibit our efforts in enriching KABs with Golog programs. In particular, we present Golog-KABs (GKABs) and define the parametric execution semantics for such framework. We also define the standard execution semantics for GKABs and call such setting S-GKABs. Last, we show in this chapter that the verification of S-GKABs is reducible to that for KABs and vice versa.
5. In Chapter 5, we extend GKABs towards Inconsistency-aware GKABs (I-GKABs). The core results presented in this chapter are the reductions of verification from various Inconsistency-aware GKABs to GKABs with standard execution semantics.

6. In Chapter 6, we extend GKABs into Context-sensitive GKABs (CSGKABs), which take into account contextual information during the evolution. We show that verification of context-sensitive temporal properties over CSGKABs can be reduced to the verification of GKABs with standard semantics. Moreover, we show that S-GKABs can be easily captured by Context-sensitive GKABs.
7. Building on the results in Chapter 5 and Chapter 6, in Chapter 7, we present Inconsistency-aware Context-Sensitive GKABs, which combine both the inconsistency handling mechanism and the contextual information. Concerning the core result, here we show that the verification of them is reducible to the verification of standard GKABs and vice versa
8. In Chapter 8, we define Alternating Golog-KABs (AGKABs), which separate the sources of non-determinism and enable fine-grained analysis over the system evolution. In this chapter, we prove that we can reduce the verification of AGKABs with various inconsistency handling mechanisms to the verification of S-GKABs and vice versa.
9. In Chapter 9, which serves as the last technical chapter in this thesis, we focus on our proposed novel framework of Semantically-Enhanced Data-Aware Processes (SEDAPs) and also show the solution for verifying SEDAPs.
10. In Chapter 10 we provide conclusions and future works.

PRELIMINARIES

This chapter introduces several preliminaries that will be used for the rest of this thesis. In particular, we briefly present the notion of relational databases, *DL-Lite* Knowledge Bases (KBs), queries, query answering over databases as well as over KB, history preserving μ -calculus ($\mu\mathcal{L}_A^{\text{EQL}}$ and $\mu\mathcal{L}_A$), and Data Centric Dynamic Systems (DCDSs). A convention on some basic notions and notations also will be presented here.

We assume some familiarities with the basic notion of Propositional Logic, and First Order Logic (FOL), for further references, please consult [167]. Moreover, in the following we make use of a countably infinite set Δ of constants.

2.1 Scenario for the Running Examples

For the running examples throughout the thesis, we consider a *furniture provider enterprise* order processing scenario as follows:

1. First, the company receives some orders.
2. Second, the company approves the orders (in case they are not yet approved).
3. The company prepares a few things that are needed for further order processing steps (such as creating the design and assigning the assembling location).
4. The assembler assembles the orders based on the given design.
5. The quality controller (QC) checks the assembled orders.
6. The delivery team delivers the orders to the delivery service.

Later on, when necessary, we will give more details on the scenario above. In some parts, we also extend this scenario as well as develop more stories based on the scenario above.

2.2 Some Basic Notions and Notations Convention

Here we briefly sketch some required basic notions and notations as follows:

Given a set A , we write $|A|$ to denote the cardinality of set A . I.e., the number of elements in the set A .

Let A and B be two arbitrary sets, as usual, a relation $f : A \times B$ over A and B is a subset of the cartesian product between A and B (i.e., $f \subseteq \{\langle a, b \rangle \mid a \in A \text{ and } b \in B\}$). The relation f is a function if for each $a \in A$ there exists exactly one $b \in B$ such that $\langle a, b \rangle \in f$. When f is a function, we sometimes write $[a \rightarrow b] \in f$ instead of $\langle a, b \rangle \in f$ to denote a tuple in f . Moreover, we say f maps a to b , written $f(a) = b$, if $[a \rightarrow b] \in f$. We write $\text{DOM}(f)$ to denote the domain of f .

Given a set V of variables, a *substitution* σ is a function $\sigma : V \rightarrow \Delta$ which maps each variable in V into a constant in Δ . Given a substitution $\sigma : V \rightarrow \Delta$, we write $x/c \in \sigma$ if $\sigma(x) = c$, i.e., σ maps x into $c \in \Delta$ (or sometimes we also say σ substitutes

x with $c \in \Delta$). We write $\sigma[x/c]$ to denote a new substitution obtained from σ such that $\sigma[x/c](x) = c$ and $\sigma[x/c](y) = \sigma(y)$ (for $y \neq x$).

We write

$$\varphi(x_1, \dots, x_n)$$

to denote an open FOL formula φ whose free variables are x_1, \dots, x_n . Given an FOL formula $\varphi(x_1, \dots, x_n)$, an FOL interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, and a substitution σ which substitutes each free variable x_i with a constant $c \in \Delta^{\mathcal{I}}$, we write

$$\varphi\sigma$$

to denote a close FOL formula obtained by applying σ to φ , i.e., substituting each free variable x_i in φ with a constant $c \in \Delta^{\mathcal{I}}$ based on σ . Moreover, we write

$$\mathcal{I} \models \varphi\sigma$$

if \mathcal{I} is a model $\varphi\sigma$.

For compactness of writing, we often write \vec{x} to denote a sequence of variables x_1, \dots, x_n . Moreover, we often say that x is a variable in \vec{x} (or we write $x \in \vec{x}$) to say that x is the variable x_i in the sequence of variables x_1, \dots, x_n (for an $i \in \{1, \dots, n\}$). Given two sequences of variables \vec{x} and \vec{y} , we write $\vec{x} \subseteq \vec{y}$, if for each variable $v \in \vec{x}$, we have $v \in \vec{y}$.

2.3 Relational Databases

We now proceed to present some preliminaries on relational databases. For further references, please consult [1, 97]. Here we make use the countably infinite set Δ of constants as the domain of the values in a database (we also call the set Δ *database domain*).

Relation Schema **Definition 2.1** (Relation Schema). A *relation schema* is simply a relation name with some arity $n > 0$. ■

Database Schema **Definition 2.2** (Database Schema). A *database schema* \mathcal{R} is a finite set $\{R_1, \dots, R_n\}$ of relation schemas. ■

Database Fact **Definition 2.3** (Database Fact). Given a relation schema R with arity n , a *database fact* (briefly *fact*) over schema R is an expression of the form $R(c_1, \dots, c_n)$ such that $c_i \in \Delta$ for $i \in \{1, \dots, n\}$. ■

Relation Instance **Definition 2.4** (Relation Instance). Given a relation schema R with arity n , a *relation instance* of R over Δ is a finite set of facts over R . ■

Database Instance **Definition 2.5** (Database Instance). Given a database schema $\mathcal{R} = \{R_1, \dots, R_n\}$, a *database instance which conforms to \mathcal{R}* is a finite set \mathbf{I} of facts over R_i for $i \in \{1, \dots, n\}$. ■

Example 2.6. Consider our running example scenario in Section 2.1, as an example of a database schema, we specify a database schema \mathcal{R} that contains the following relation schemas

- ORDER(id, name, processing_status, customerid, designer, assembler, quality_controller, assembling_loc, design)
- DELIVERED_ORDER(id, delivery_date)

Essentially, the database schema \mathcal{R} contains two relation schema, namely ORDER and DELIVERED_ORDER, where the former stores the information about customer orders and the latter stores the information about orders that has been delivered.

An example of a database instance which conforms to \mathcal{R} is as follows:

$$\mathbf{I} = \{\text{ORDER}(123, \text{chair}, \text{received}, 456, \text{NULL}, \text{NULL}, \text{NULL}, \text{NULL}, \text{ecodesign}), \\ \text{ORDER}(321, \text{table}, \text{approved}, 654, \text{NULL}, \text{NULL}, \text{NULL}, \text{NULL}, \text{NULL})\}.$$

Definition 2.7 (Active Domain of a Database Instance). Given a database instance \mathbf{I} , the *active domain* of \mathbf{I} is the smallest set $\text{ADOM}(\mathbf{I}) \subseteq \Delta$ of constants such that for each $R(c_1, \dots, c_n) \in \mathbf{I}$, we have $c_i \in \text{ADOM}(\mathbf{I})$ for $i \in \{1, \dots, n\}$. ■

Active Domain of a Database Instance

Intuitively, an active domain of a database instance \mathbf{I} is a finite set of constants that explicitly present in \mathbf{I} .

Definition 2.8 (FOL Interpretation Obtained From a Database). Given a database instance \mathbf{I} which conforms to a database schema $\mathcal{R} = \{R_1, \dots, R_n\}$, we define an *FOL Interpretation obtained from \mathbf{I}* as a usual FOL Interpretation $\mathcal{I}_{\mathbf{I}} = (\Delta^{\mathcal{I}_{\mathbf{I}}}, \mathcal{I}_{\mathbf{I}})$ such that

FOL Interpretation Obtained From a Database

1. $\Delta^{\mathcal{I}_{\mathbf{I}}} = \Delta$,
2. $R_i^{\mathcal{I}_{\mathbf{I}}} = \{\langle c_1, \dots, c_m \rangle \mid R_i(c_1, \dots, c_m) \in \mathbf{I} \text{ for } i \in \{1, \dots, n\}\}.$

■

To simplify the notation, when it is clear from the context, we often just write \mathbf{I} to denote $\mathcal{I}_{\mathbf{I}}$. For example, given a close FOL formula φ , we write $\mathbf{I} \models \varphi$ to denote $\mathcal{I}_{\mathbf{I}} \models \varphi$. Informally speaking, in this case we want to consider a database instance simply as an FOL interpretation.

2.4 DL-Lite Knowledge Bases

In this thesis we use Description Logic (DL) [16] to express the Knowledge Bases (KB). In particular, we resort to a specific DL family, namely *DL-Lite* [50, 11], which is specifically tuned to have low complexity of reasoning while still expressive enough to capture the domain of interest. In this section, we briefly sketch some members of *DL-Lite* family, namely *DL-Lite_A* [151, 53], and *DL-Lite_R* [50].

DL-Lite_A. *DL-Lite_A* distinguishes the set of constants into the set of *objects* and the set of *values*. Hence, for this section only, we assume that the set Δ consists of a set Δ_O of objects and a set Δ_V of values such that $\Delta = \Delta_O \uplus \Delta_V$. The *DL-Lite_A* allows for expressing (i) *concepts*, representing sets of objects, (ii) *roles*, representing binary

relations between objects, and (iii) *attributes*, representing binary relations between objects and values. The syntax of $DL\text{-}Lite_A$ expressions is given below.

Syntax of $DL\text{-}Lite_A$

Definition 2.9 (The Syntax of $DL\text{-}Lite_A$ Expressions). The syntax of $DL\text{-}Lite_A$ expressions (concept, role and attribute) is defined by the following grammar:

$$B ::= N \mid \exists R \mid \delta(U) \quad R ::= P \mid P^-$$

where

- N , P , and U respectively denote a *concept name*, a *role name*, and an *attribute name*,
- P^- denotes the *inverse of a role*,
- B and R respectively denote *concepts* and *roles*.
- The concept $\exists R$, also called *unqualified existential restriction*, denotes the *domain* of a role R , i.e., the set of objects that R relates to some object.
- Similarly, the concept $\delta(U)$ denotes the *attribute domain* of U , i.e., the set of objects that U relates to some value.

■

We assume that the set of concept, role, and attribute names are disjoint.

To formally present the definition of $DL\text{-}Lite_A$ KBs, we first explain the notion of $DL\text{-}Lite_A$ ABox and TBox as follows:

$DL\text{-}Lite_A$ ABox

Definition 2.10 ($DL\text{-}Lite_A$ ABox). A $DL\text{-}Lite_A$ ABox is a finite set A of ABox assertions of the form

$$N(o_1), \quad P(o_1, o_2), \quad U(o_1, v),$$

where $o_1, o_2 \in \Delta_O$ denote objects and $v \in \Delta_V$ denotes a value. Consecutively, from left to right, $N(o_1)$ is called *concept assertion*, $P(o_1, o_2)$ is called *role assertion*, and $U(o_1, v)$ is called *attribute assertion*. ■

The notions of *entailment*, *satisfaction*, and *model* of an ABox is as usual [52]. Informally, an ABox can be considered as a data storage. Notice that we can also view an ABox as a database instance where the schema is the set of concept, role, and attribute names. Similar to Definition 2.7, we define an *active domain* of ABox A , denoted by $\text{ADOM}(A)$, as the set of constants from Δ (i.e., objects and values) that explicitly present in A .

Active Domain
of an ABox

Definition 2.11 (Active Domain of an ABox). Given an ABox A , an *active domain* of A is a finite set $\text{ADOM}(A) \subseteq \Delta$ of constants constructed as follows

1. For each concept assertion $N(o) \in A$, we have $o \in \text{ADOM}(A)$,
2. For each role assertion $P(o_1, o_2) \in A$, we have $o_1, o_2 \in \text{ADOM}(A)$,
3. For each attribute assertion $U(o, v) \in A$, we have $o, v \in \text{ADOM}(A)$.

■

We now proceed to explain the notion of TBox that intuitively encodes the domain knowledge as follows.

Definition 2.12 (*DL-Lite_A TBox*). A *DL-Lite_A TBox* is a finite set

DL-Lite_A TBox

$$T = T_p \uplus T_n \uplus T_f,$$

with

- T_p a finite set of *positive inclusion assertions* of the form

$$B_1 \sqsubseteq B_2, \quad R_1 \sqsubseteq R_2, \quad U_1 \sqsubseteq U_2.$$

- T_n a finite set of *negative inclusion assertions* of the form

$$B_1 \sqsubseteq \neg B_2, \quad R_1 \sqsubseteq \neg R_2, \quad U_1 \sqsubseteq \neg U_2.$$

- T_f a finite set of *functionality assertions* of the form

$$(\text{funct } R), \quad (\text{funct } U).$$

where (i) Each B_i and R_i respectively denote *concepts* and *roles*. (ii) U denotes an *attribute name*. Additionally, as usual in *DL-Lite_A TBoxes*, we impose that roles and attributes occurring in functionality assertions cannot be specialized (i.e., they cannot occur in the right-hand side of positive inclusions). ■

The notions of *entailment*, *satisfaction*, and *model* of a TBox is as usual [52].

We call the set of concept, role, and attribute names that appear in TBox T a *vocabulary of TBox T* , denoted by $\text{voc}(T)$. W.l.o.g. given a TBox T , we assume that $\text{voc}(T)$ contains all possible concept, role, and attribute names. Notice that we can simply add an assertion $N \sqsubseteq N$ (resp. $P \sqsubseteq P$ and $U \sqsubseteq U$) into the TBox T in order to add a concept name N (resp. role name P , and attribute name U) into $\text{voc}(T)$ such that $\text{voc}(T)$ contains all possible concept, role, and attribute names, and without changing the expected set of models of the TBox T (hence, preserving the deductive closures of T). Moreover, we call an ABox A *is over* $\text{voc}(T)$ if it consists of ABox assertions of the form either $N(o_1)$, $P(o_1, o_2)$, or $U(o_1, v)$, where $N, P, U \in \text{voc}(T)$.

TBox Vocabulary

Having the notion of the *DL-Lite_A TBox* and ABox in place, we are ready to introduce a *DL-Lite_A Knowledge Bases (KB)* as follows:

Definition 2.13 (*DL-Lite_A Knowledge Base*). A *DL-Lite_A KB* is a pair $\langle T, A \rangle$, where (i) T is a *DL-Lite_A TBox*, (ii) A is a *DL-Lite_A ABox* over $\text{voc}(T)$ ■

DL-Lite_A Knowledge Base

For the semantics of *DL-Lite_A* expressions, as in [52], we adopt the standard FOL semantics of DLs based on FOL interpretations. We also interpret objects as well as values over distinct domains. Additionally, we adopt the *standard name assumption*, i.e.,

1. *Unique name assumption* holds (different constants denote different objects or values).
2. The interpretation domain contains the set of objects and values and each value (as well as object) is interpreted as itself.

Definition 2.14 (*Semantics of DL-Lite_A Expressions*). The semantics of *DL-Lite_A* expressions is given by an interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ where:

Semantics of DL-Lite_A

- $\Delta^{\mathcal{I}} = \Delta_O \uplus \Delta_V$ is an *interpretation domain*.

Role name	P	$P^{\mathcal{I}} \subseteq \Delta_O^{\mathcal{I}} \times \Delta_O^{\mathcal{I}}$
Inverse of a role	P^{-}	$(P^{-})^{\mathcal{I}} = \{\langle o, o' \rangle \mid \langle o', o \rangle \in P^{\mathcal{I}}\}$
Attribute name	U	$U^{\mathcal{I}} \subseteq \Delta_O^{\mathcal{I}} \times \Delta_V^{\mathcal{I}}$
Concept name	N	$N^{\mathcal{I}} \subseteq \Delta_O^{\mathcal{I}}$
Unqualified existential restriction	$\exists R$	$(\exists R)^{\mathcal{I}} = \{o \mid \exists o'. \langle o, o' \rangle \in R^{\mathcal{I}}\}$
Domain of an attribute	$\delta(U)$	$\delta(U)^{\mathcal{I}} = \{o \mid \exists v. \langle o, v \rangle \in U^{\mathcal{I}}\}$

Table 1: Semantics $DL\text{-}Lite_{\mathcal{A}}$ expressions

- $\cdot^{\mathcal{I}}$ is an *interpretation function* such that
 - for each object $o \in \Delta_O$, $o^{\mathcal{I}} = o$,
 - for each value $v \in \Delta_V$, $v^{\mathcal{I}} = v$,
 - the conditions in Table 1 are satisfied.

■

The semantics of $DL\text{-}Lite_{\mathcal{A}}$ ABox is defined as follows:

*Semantics of
 $DL\text{-}Lite_{\mathcal{A}}$ ABox*

Definition 2.15 (Semantics of $DL\text{-}Lite_{\mathcal{A}}$ ABox). An interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ satisfies an ABox assertion:

$$\begin{aligned}
 A(o) & \quad \text{if} \quad o^{\mathcal{I}} \in A^{\mathcal{I}}; \\
 P(o_1, o_2) & \quad \text{if} \quad \langle o_1^{\mathcal{I}}, o_2^{\mathcal{I}} \rangle \in P^{\mathcal{I}}; \\
 U(o, v) & \quad \text{if} \quad \langle o^{\mathcal{I}}, v^{\mathcal{I}} \rangle \in U^{\mathcal{I}}.
 \end{aligned}$$

■

The semantics of $DL\text{-}Lite_{\mathcal{A}}$ TBox is defined as follows:

*Semantics of
 $DL\text{-}Lite_{\mathcal{A}}$ TBox*

Definition 2.16 (Semantics of $DL\text{-}Lite_{\mathcal{A}}$ TBox). An interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ satisfies a TBox assertion:

$$\begin{aligned}
 B_1 \sqsubseteq B_2 & \quad \text{if} \quad B_1^{\mathcal{I}} \subseteq B_2^{\mathcal{I}}; \\
 R_1 \sqsubseteq R_2 & \quad \text{if} \quad R_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}; \\
 U_1 \sqsubseteq U_2 & \quad \text{if} \quad U_1^{\mathcal{I}} \subseteq U_2^{\mathcal{I}}; \\
 B_1 \sqsubseteq \neg B_2 & \quad \text{if} \quad \forall o. o \in B_1^{\mathcal{I}} \rightarrow o \notin B_2^{\mathcal{I}}; \\
 R_1 \sqsubseteq \neg R_2 & \quad \text{if} \quad \forall o, o'. \langle o, o' \rangle \in R_1^{\mathcal{I}} \rightarrow \langle o, o' \rangle \notin R_2^{\mathcal{I}}; \\
 U_1 \sqsubseteq \neg U_2 & \quad \text{if} \quad \forall o, v. \langle o, v \rangle \in U_1^{\mathcal{I}} \rightarrow \langle o, v \rangle \notin U_2^{\mathcal{I}}; \\
 (\text{funct } R) & \quad \text{if} \quad \forall o, o', o''. \langle o, o' \rangle \in R^{\mathcal{I}} \wedge \langle o, o'' \rangle \in R^{\mathcal{I}} \rightarrow o' = o''; \\
 (\text{funct } U) & \quad \text{if} \quad \forall o, v', v''. \langle o, v' \rangle \in R^{\mathcal{I}} \wedge \langle o, v'' \rangle \in R^{\mathcal{I}} \rightarrow v' = v'';
 \end{aligned}$$

■

The notions of *entailment*, *satisfaction*, and *model* are as usual (c.f. [52]). An interpretation \mathcal{I} *satisfies a TBox* T , written $\mathcal{I} \models T$, if \mathcal{I} satisfies all TBox assertions in T . Similarly, an interpretation \mathcal{I} *satisfies an ABox* A , written $\mathcal{I} \models A$, if \mathcal{I} satisfies all ABox assertions in A . Furthermore, an interpretation \mathcal{I} *satisfies a KB* $\langle T, A \rangle$,

written $\mathcal{I} \models \langle T, A \rangle$, if \mathcal{I} satisfies the TBox T and the ABox A . We say A is *T-consistent* if $\langle T, A \rangle$ is satisfiable, i.e., admits at least one model, otherwise we say A is *T-inconsistent*. Additionally, in this thesis we assume that, given a TBox T , all concepts and roles in $\text{VOC}(T)$ are satisfiable, i.e., for every concept N in T , there exists at least one model \mathcal{I} of T such that $N^{\mathcal{I}}$ is non-empty, and similarly for roles.

As an observation, notice that in *DL-Lite_A*, positive inclusion assertions alone cannot generate inconsistency. This is an immediate consequence of Lemma 4.5 in [53] which essentially says that we can always find a model for a *DL-Lite_A* KB $\langle T_p, A \rangle$ where T_p only contains positive inclusion assertions. However, positive inclusion assertions might involve in causing inconsistency by interacting with negative inclusions.

Example 2.17. Continuing our running example using the scenario in Section 2.1. We specify a *DL-Lite_A* KB $\langle T, A \rangle$, where the TBox T contains the following assertions:

ApprovedOrder \sqsubseteq Order	$\exists \text{assembledBy}^- \sqsubseteq$ Employee
AssembledOrder \sqsubseteq Order	$\exists \text{assembledBy} \sqsubseteq$ Order
DeliveredOrder \sqsubseteq Order	$\exists \text{designedBy}^- \sqsubseteq$ Employee
ReceivedOrder \sqsubseteq Order	$\exists \text{designedBy} \sqsubseteq$ Order
Designer \sqsubseteq Employee	$\exists \text{checkedBy}^- \sqsubseteq$ Employee
Assembler \sqsubseteq Employee	$\exists \text{checkedBy} \sqsubseteq$ Order
QualityController \sqsubseteq Employee	$\exists \text{hasAssemblingLoc}^- \sqsubseteq$ Location
Designer $\sqsubseteq \neg \text{Assembler}$	$\exists \text{hasAssemblingLoc} \sqsubseteq$ Order
Designer $\sqsubseteq \neg \text{QualityController}$	$\exists \text{hasDesign}^- \sqsubseteq$ Design
Assembler $\sqsubseteq \neg \text{QualityController}$	$\exists \text{hasDesign} \sqsubseteq$ Order
(funct hasAssemblingLoc)	
(funct hasDesign)	

The intuition of some TBox assertions presented above are as follows:

- The assertion $\text{ApprovedOrder} \sqsubseteq \text{Order}$ states that every approved order is an order.
- The assertion $\text{Designer} \sqsubseteq \neg \text{Assembler}$ encodes a constraint that a designer is not an assembler.
- The assertion $\exists \text{hasAssemblingLoc} \sqsubseteq \text{Order}$ states that those that can have an assembling location must be an order.
- The assertion $\exists \text{checkedBy}^- \sqsubseteq \text{Employee}$ says that something can be only checked by an employee.
- The assertion (funct hasAssemblingLoc) says that the role hasAssemblingLoc is functional. Informally, it constraints the domain of hasAssemblingLoc to have only a single range.

Moreover, the ABox A is specified as follows:

$$A = \{\text{ReceivedOrder}(\text{chair}), \text{ApprovedOrder}(\text{table})\}.$$

Intuitively, the ABox A states the fact that there are a received order of chair and an approved order of table.

DL-Lite_R. *DL-Lite_R* is a sublanguage of *DL-Lite_A* which is obtained by dropping functionality assertions. *DL-Lite_R* is also the basis of OWL 2 QL (a profile¹ of the Web Ontology Language OWL 2 standardized by the W3C.). In W3C terminology, a *profile* is a sublanguage of the full OWL 2, defined by suitable syntactic restrictions. OWL 2 QL is specifically designed for building an ontology layer to wrap possibly very large data sources. Notably, it allows for query answering over ontologies with the same data complexity as plain SQL query evaluation over relational databases.

2.5 Query Answering

Roughly speaking, queries are expressions to ask information, and query answering is a mechanism to extract some information (called *answer*) from an information source. In the following, we introduce various kind of queries that will be used later, and also explain how the answers are computed when we pose such queries over an information/data source.

As a start, we introduce a very expressive query, namely FOL queries, as follows:

FOL Query **Definition 2.18** (FOL Query). An *FOL query* Q is an FOL formula without function symbols that might use some constants in Δ . ■

As usual, given an FOL query Q , we call Q a *boolean query* or a *closed query* if Q has no free variables, otherwise we call it an *open query*.

2.5.1 Query Answering Over Databases

We use queries to formulate questions to be asked over a database. Given a database instance \mathbf{I} and a query q , query answering is aiming to obtain the answers to the query q which are formed by elements of Δ . In the following we introduce some query languages that will be used later to query a database and also the notion of answers to a query. For further references, please consult [1, 97].

Given an FOL query φ , and a database instance \mathbf{I} which conforms to a database schema \mathcal{R} , we say that the FOL query φ is *over \mathcal{R} and \mathbf{I}* if the atoms in φ are made from relation schemas in \mathcal{R} and might use some constants in $\text{ADOM}(\mathbf{I})$.

Now, we present an interesting fragment of FOL queries which will be used quite often later, namely Union of Conjunctive Query (UCQ).

Union of Conjunctive Query (UCQ) **Definition 2.19** (Union of Conjunctive Query Over a Database). Given a database instance \mathbf{I} which conforms to a database schema \mathcal{R} , a *Union of Conjunctive Query (UCQ)* q over \mathcal{R} and \mathbf{I} is an FOL query of the form

$$\exists \vec{y}_1. \text{conj}_1(\vec{x}, \vec{y}_1) \vee \cdots \vee \exists \vec{y}_n. \text{conj}_n(\vec{x}, \vec{y}_n),$$

where each $\text{conj}_i(\vec{x}, \vec{y}_i)$ in q is a conjunction of atoms whose predicates are either

- relation schemas in \mathcal{R} , or
- equality assertions

which involve free variables \vec{x} and existentially quantified variables $\vec{y}_1, \dots, \vec{y}_n$, and/or elements of $\text{ADOM}(\mathbf{I})$. ■

¹ In W3C terminology, a *profile* is a sublanguage defined by suitable syntactic restrictions.

Example 2.20. Continuing our running example, recall the database schema \mathcal{R} and the database instance \mathbf{I} in Example 2.6, an example of a UCQ q over \mathcal{R} and \mathbf{I} is

$$\exists x_2 x_4 x_5 x_6 x_7 x_8 x_9. \text{ORDER}(x_1, x_2, \text{"received"}, x_4, \dots, x_9)$$

that retrieves the id of received orders.

We sometimes say a UCQ q is over \mathcal{R} to refer to a UCQ whose atoms are made using relation schemas in \mathcal{R} .

Next, we explain the notion of answers to a query Q over a database instance \mathbf{I} as a result of evaluating Q over \mathbf{I} as follows.

Definition 2.21 (Answers to a Query). Given a database instance \mathbf{I} which conforms to a database schema \mathcal{R} and an FOL query Q over a database schema \mathcal{R} . The *answers to Q over \mathbf{I}* is the set $\text{ANS}(Q, \mathbf{I})$ of substitutions, in which each substitution $\sigma \in \text{ANS}(Q, \mathbf{I})$ substitutes the free variables of Q with elements of Δ such that $\mathbf{I} \models Q\sigma$. If Q is a boolean query then its answer is either a singleton set of an empty substitution (corresponding to **true**) or an empty set (corresponding to **false**). ■

Answers to a Query

As customary, we can view each substitution σ (which is an answer to a query $Q(x_1, \dots, x_n)$) simply as a tuple of elements of Δ , assuming some ordering of the free variables x_1, \dots, x_n of Q . Therefore, given a query $Q(x_1, \dots, x_n)$ and a substitution $\sigma \in \text{ANS}(Q(x_1, \dots, x_n), \mathbf{I})$ such that $\sigma(x_i) = c_i$ (for $1 \leq i \leq n$), we sometimes also write it as either $\langle c_1, \dots, c_n \rangle \in \text{ANS}(Q(x_1, \dots, x_n), \mathbf{I})$ or $\vec{c} \in \text{ANS}(Q(\vec{x}), \mathbf{I})$.

Example 2.22. Continuing Example 2.20, recall the query q as follows:

$$\exists x_2 x_4 x_5 x_6 x_7 x_8 x_9. \text{ORDER}(x_1, x_2, \text{"received"}, x_4, \dots, x_9)$$

a substitution σ that substitutes x_1 to “123” is an answer to q over \mathbf{I} (i.e., $\sigma \in \text{ANS}(q, \mathbf{I})$).

Another interesting fragment of FOL query that will be used later is *Domain Independent FOL (DI-FOL) query*. To formally explain the notion of DI-FOL query, several preliminaries need to be introduced as follows.

Definition 2.23 (FOL Interpretation Obtained From a Database w.r.t. a Certain Domain). Given a set Δ' of constants such that $\Delta' \subseteq \Delta$, and a database instance \mathbf{I} which conforms to a database schema $\mathcal{R} = \{R_1, \dots, R_n\}$, and $\text{ADOM}(\mathbf{I}) \subseteq \Delta'$, we define an *FOL Interpretation obtained from \mathbf{I} w.r.t. Δ'* as a usual FOL Interpretation $\mathcal{I}_{\mathbf{I}}^{\Delta'} = (\Delta^{\mathcal{I}_{\mathbf{I}}^{\Delta'}}, \mathcal{I}_{\mathbf{I}}^{\Delta'})$ such that

FOL Interpretation Obtained From a Database w.r.t. a Certain Domain

- $\Delta^{\mathcal{I}_{\mathbf{I}}^{\Delta'}} = \Delta'$,
- $R_i^{\mathcal{I}_{\mathbf{I}}^{\Delta'}} = \{\langle c_1, \dots, c_m \rangle \mid R_i(c_1, \dots, c_m) \in \mathbf{I}\}$ for $i \in \{1, \dots, n\}$.

■

When it is clear from the context, we often just write $\mathbf{I}^{\Delta'}$ to denote $\mathcal{I}_{\mathbf{I}}^{\Delta'}$. For example, given a close FOL query Q , we write $\mathbf{I}^{\Delta'} \models Q$ if $\mathcal{I}_{\mathbf{I}}^{\Delta'} \models Q$.

Answers of a Query
w.r.t. a Certain
Domain

Definition 2.24 (Answers of a Query w.r.t. a Certain Domain). Given a set Δ' of constants such that $\Delta' \subseteq \Delta$, an FOL query $Q(x_1, \dots, x_n)$ over a database schema \mathcal{R} , a database instance \mathbf{I} which conforms to a database schema \mathcal{R} , and $\text{ADOM}(\mathbf{I}) \subseteq \Delta'$. The *answers to Q over \mathbf{I} w.r.t. Δ'* is the set $\text{ANS}_{\Delta'}(Q, \mathbf{I})$ of substitutions, in which each substitution $\sigma \in \text{ANS}_{\Delta'}(Q, \mathbf{I})$ substitutes the free variables of Q with elements of Δ' such that $\mathbf{I}^{\Delta'} \models Q\sigma$. If Q is a boolean query then its answer is either a singleton set of an empty substitution (corresponding to true) or an empty set (corresponding to false). ■

Having the required preliminaries in hand, we are ready to present the definition of DI-FOL query as follows.

Domain Independent
FOL Query

Definition 2.25 (Domain Independent FOL Query). An FOL query Q is a *Domain Independent FOL (DI-FOL) query* if for every database instance \mathbf{I} which conforms to a database schema \mathcal{R} , and for every database domain Δ' and Δ'' such that $\text{ADOM}(\mathbf{I}) \subseteq \Delta' \subseteq \Delta$ and $\text{ADOM}(\mathbf{I}) \subseteq \Delta'' \subseteq \Delta$, we have

$$\text{ANS}_{\Delta'}(Q, \mathbf{I}) = \text{ANS}_{\Delta''}(Q, \mathbf{I})$$

■

The definition above intuitively said that given a database instance \mathbf{I} and a query Q , the answers to Q over \mathbf{I} are the same no matter whether the domain of database values is the $\text{ADOM}(\mathbf{I})$, the whole set Δ of constants, or something between $\text{ADOM}(\mathbf{I})$ and Δ (i.e., the answers to the query is independent from the domain of database values that we use).

Example 2.26. The query in our previous running example (i.e., Examples 2.20 and 2.22)

$$\exists x_2 x_4 x_5 x_6 x_7 x_8 x_9. \text{ORDER}(x_1, x_2, \text{"received"}, x_4, \dots, x_9)$$

is a domain independent query. On the other hand, the following query

$$\neg \exists x_2 x_4 x_5 x_6 x_7 x_8 x_9. \text{ORDER}(x_1, x_2, \text{"received"}, x_4, \dots, x_9)$$

is an example of a query that is not domain independent.

2.5.2 Query Answering Over Knowledge Bases

We use queries to access KBs and extract values (constants) of interest, i.e., to ask some questions over a KB and obtain answers. As in Section 2.4, in the following we assume that the set Δ consists of a set Δ_O of objects and a set Δ_V of values such that $\Delta = \Delta_O \uplus \Delta_V$. Below, we introduce a particular form of queries that we will use later to query KBs, namely union of conjunctive queries over KB:

Union of Conjunctive
Query Over a KB

Definition 2.27 (Union of Conjunctive Query Over a KB). Given a KB $\langle T, A \rangle$, a *Union of Conjunctive Query (UCQ) q over $\langle T, A \rangle$* is an FOL query of the form

$$\exists \vec{y}_1. \text{conj}_1(\vec{x}, \vec{y}_1) \vee \dots \vee \exists \vec{y}_n. \text{conj}_n(\vec{x}, \vec{y}_n),$$

where each $\text{conj}_i(\vec{x}, \vec{y}_i)$ in q is a conjunction of atoms whose predicates are either

- concept/role/attribute names in $\text{VOC}(T)$, or
- equality assertions

which involve free variables \vec{x} and existentially quantified variables $\vec{y}_1, \dots, \vec{y}_n$, and/or elements of $\text{ADOM}(A)$. ■

We sometimes say a UCQ q is over T to refer to a UCQ whose atoms are made using concept/role/attribute names in $\text{VOC}(T)$.

Example 2.28. Recall the TBox T and the ABox A specified in Example 2.17, an example of UCQ q over $\langle T, A \rangle$ is:

$$\text{ApprovedOrder}(x) \vee (\text{AssembledOrder}(x) \wedge \exists y. \text{checkedBy}(x, y))$$

that retrieves either approved orders or assembled orders that has been checked by someone.

In the setting of query answering over KBs, we are interested to the answers which are “true” in every model of the corresponding KB. Such answers are called certain answers and the definition is as follows:

Definition 2.29 (Certain Answer of UCQs). Given a KB $\langle T, A \rangle$ and a UCQ q over $\langle T, A \rangle$, the (*certain*) *answers* to UCQ q over a KB $\langle T, A \rangle$ is the set $\text{cert}(q, T, A)$ of substitutions σ of the free variables of q with elements of Δ such that $\mathcal{I} \models q\sigma$ for every model \mathcal{I} of $\langle T, A \rangle$ (i.e., $q\sigma$ evaluates to true in every model of $\langle T, A \rangle$). If q is a boolean query then its *certain answer* is either a singleton set of an empty substitution (corresponding to true) or an empty set (corresponding to false). ■

Certain Answer of UCQs

Similar to answers to a query over a database instance (c.f. Definition 2.21), we can view each substitution $\sigma \in \text{cert}(q, T, A)$ simply as a tuple of elements of Δ , assuming some ordering of the free variables of q . I.e., given a query $q(x_1, \dots, x_n)$ and a substitution $\sigma \in \text{cert}(q(x_1, \dots, x_n), T, A)$ such that $\sigma(x_i) = c_i$ (for $1 \leq i \leq n$), we sometimes also write it as either $\langle c_1, \dots, c_n \rangle \in \text{cert}(q(x_1, \dots, x_n), T, A)$ or $\vec{c} \in \text{cert}(q(\vec{x}), T, A)$.

Example 2.30. Continuing Example 2.28, recall the query q as follows: $\text{ApprovedOrder}(x)$. A substitution σ that substitutes x to “table” is an answer to q over A (i.e., $\sigma \in \text{cert}(q, T, A)$).

In this work, we also consider the extension of UCQs named *EQL-Lite*(UCQ) [51] (briefly, ECQs), which is an FOL query whose atoms are UCQs evaluated according to the certain answer semantics above (see Definition 2.29).

Definition 2.31 (EQL-Lite(UCQ)). Formally, given a KB $\langle T, A \rangle$, an *ECQ* over $\langle T, A \rangle$ is a (possibly open) formula of the form:

EQL-Lite(UCQ)

$$Q ::= [q] \mid \neg Q \mid Q_1 \wedge Q_2 \mid \exists x. Q$$

where q is an UCQ over $\langle T, A \rangle$, and $[q]$ denotes the fact that q is evaluated under the (minimal) knowledge operator [51] (We often omit the square brackets for single-atom UCQs). We call *epistemic atoms* the formula $[q]$ occurring in an ECQ. ■

Example 2.32. Recall the TBox T and the ABox A specified in Example 2.17, an example of ECQ Q over $\langle T, A \rangle$ is:

$$\exists x. [\text{Order}(x)] \wedge \neg [\text{DeliveredOrder}(x)]$$

that checks whether there exists an order that is not yet delivered.

To explain how to compute certain answers to ECQ, we need to introduce several preliminaries as follows:

FOL Query of an ECQ Query

Definition 2.33 (FOL Query Obtained From an ECQ Query). Given an ECQ query $Q(\vec{x})$ with epistemic atoms $[q_1], \dots, [q_n]$. We define an *FOL query obtained from an ECQ Q* (briefly FOL query of Q), denoted by $Q_{FOL}(\vec{x})$, as query obtained from Q by replacing each epistemic atom $[q_i]$ with a new predicate R_{q_i} where the arity of R_{q_i} is the number of free variables in $[q_i]$. ■

FOL Interpretation for an FOL Query of an ECQ

Definition 2.34 (FOL Interpretation for an FOL Query of an ECQ). Given a KB $\langle T, A \rangle$, and an ECQ $Q(\vec{x})$ with epistemic atoms $[q_1], \dots, [q_n]$. Let $Q_{FOL}(\vec{x})$ be an FOL query obtained from $Q(\vec{x})$. We define an *FOL interpretation w.r.t. $\langle T, A \rangle$ and $Q(\vec{x})$* as an FOL interpretation $\mathcal{I}_{Q, \langle T, A \rangle} = (\Delta^{\mathcal{I}_{Q, \langle T, A \rangle}}, \mathcal{I}_{Q, \langle T, A \rangle})$ such that

- $\Delta^{\mathcal{I}_{Q, \langle T, A \rangle}} = \Delta$
- for every predicates $R_{q_i}(x_1, \dots, x_n)$, we have

$$R_{q_i}^{\mathcal{I}_{Q, \langle T, A \rangle}} = \{ \langle c_1, \dots, c_n \rangle \mid \sigma \in \text{cert}(q_i(x_1, \dots, x_n), T, A) \text{ and } x_i/c_i \in \sigma \}$$

■

Certain Answers of an ECQ

Having the machinery in hand, we are now ready to explain how to compute the certain answers of an ECQ. Given a KB $\langle T, A \rangle$, an ECQ $Q(\vec{x})$, the *certain answers of $Q(\vec{x})$ over $\langle T, A \rangle$* , denoted by $\text{CERT}(Q(\vec{x}), T, A)$, is a set of substitutions σ of the free variables of $Q(\vec{x})$ with elements of Δ such that $\mathcal{I}_{Q, \langle T, A \rangle} \models Q_{FOL}(\vec{x})\sigma$. If Q is a boolean query then its *certain answer* is either a singleton set of an empty substitution (corresponding to true) or an empty set (corresponding to false). Intuitively, the *certain answers* $\text{CERT}(Q, T, A)$ of an ECQ Q over $\langle T, A \rangle$ are obtained by computing the certain answers of the UCQs embedded in Q , then composing such answers through the FO constructs in Q .

Now we introduce an interesting fragment of ECQ namely domain independent ECQ.

Domain Independent ECQ

Definition 2.35 (Domain Independent ECQ (DI-ECQ)). Given an ECQ Q with epistemic atoms $[q_1], \dots, [q_n]$, we say Q is a *Domain Independent ECQ (DI-ECQ)*, if for each FOL interpretation $\mathcal{I}_1 = (\Delta^{\mathcal{I}_1}, \mathcal{I}_1)$ and $\mathcal{I}_2 = (\Delta^{\mathcal{I}_2}, \mathcal{I}_2)$ for Q_{FOL} such that $\Delta^{\mathcal{I}_1} \subseteq \Delta$, $\Delta^{\mathcal{I}_2} \subseteq \Delta$, and $R_{q_i}^{\mathcal{I}_1} = R_{q_i}^{\mathcal{I}_2}$ for all atomic relations R_{q_i} , we have that

$$\mathcal{I}_1 \models Q_{FOL}\sigma \text{ if and only if } \mathcal{I}_2 \models Q_{FOL}\sigma$$

for any substitution σ . ■

Example 2.36. The ECQ in Example 2.32 is a DI-ECQ.

2.5.2.1 FO-Rewritability of *DL-Lite*

Similar to Definition 2.8, considering that an ABox is also a set of facts, we can define an FOL interpretation obtained from ABox A as follows:

Definition 2.37 (FOL Interpretation Obtained From an ABox). Given a KB $\langle T, A \rangle$, we define an *FOL Interpretation obtained from A* as a usual FOL Interpretation $\mathcal{I}_A = (\Delta^{\mathcal{I}_A}, \cdot^{\mathcal{I}_A})$ such that

- $\Delta^{\mathcal{I}_A} = \Delta$ (Note that Δ contains both objects and values),
- for every concept name $N \in \text{voc}(T)$, $N^{\mathcal{I}_A} = \{o \mid N(o) \in A\}$.
- for every role name $P \in \text{voc}(T)$, $P^{\mathcal{I}_A} = \{\langle o_1, o_2 \rangle \mid P(o_1, o_2) \in A\}$.
- for every attribute name $U \in \text{voc}(T)$, $U^{\mathcal{I}_A} = \{\langle o, v \rangle \mid U(o, v) \in A\}$.

■

To simplify the notation, when it is clear from the context, we often just write A to denote \mathcal{I}_A . For example, given a close UCQ q , we write $A \models q$ to denote $\mathcal{I}_A \models q$. Informally, we want to consider an ABox simply as an FOL interpretation. Furthermore, having Definition 2.37 in hand, we can also define an evaluation of query Q over an ABox A (similar to Definition 2.21) as follows.

Definition 2.38 (Query Evaluation Over an ABox). Given a KB $\langle T, A \rangle$ and an FOL query Q over $\langle T, A \rangle$. We define *answers to Q over A* as a set $\text{ANS}(Q, A)$ of substitutions σ of the free variables of q with elements of Δ such that $A \models Q\sigma$. As before, If Q is a boolean query then its *answer* is either a singleton set of an empty substitution (corresponding to **true**) or an empty set (corresponding to **false**). ■

We now recall that DL-Lite enjoys the *FO rewritability* property, which means as follows.

Theorem 2.39 (FO rewritability of *DL-Lite_A* [52]). *Given a KB $\langle T, A \rangle$ and a UCQ q , we have*

$$\text{cert}(q, T, A) = \text{ANS}(\text{rew}(q, T), A),$$

where $\text{rew}(q, T)$ is a UCQ computed by the query rewriting algorithm in [52]. □

Theorem 2.39 intuitively said that to compute the certain answers to a UCQ q over a KB $\langle T, A \rangle$, we can rewrite the query q to compile away the TBox T as well as incorporate the domain knowledge encoded in T into q , and then we can just evaluate the rewritten query $\text{rew}(q, T)$ over the ABox A . As in [53], such query rewriting algorithm is called *the perfect reformulation algorithm*.

Furthermore, the perfect reformulation algorithm above can be extended to ECQs as in [51], and that its effect is to “compile away” the TBox. Precisely this statement is stated below.

Theorem 2.40 (FO Rewritability of Answering ECQ[51]). *Given a KB $\langle T, A \rangle$, and an ECQ Q , we have*

$$\text{CERT}(Q, T, A) = \text{ANS}(\text{rew}(Q, T), A),$$

Where $\text{rew}(Q, T)$ is an FOL query. Furthermore, if Q is DI-ECQ query, then $\text{rew}(Q, T)$ is DI-FOL query. \square

2.5.2.2 Consistency Check via Query Answering

We recall that checking the satisfiability of a $DL\text{-}Lite_A$ KB $\langle T, A \rangle$ is FO rewritable, i.e., it can be reduced to evaluating a boolean FOL query over A [50]. For brevity of the notation that we will use later, we introduce several abbreviations below:

Abbreviations For
Query Atom

Definition 2.41 (Abbreviations For Query Atom). We define some notations to compactly express various atoms in a query as follows:

- An atom $B(x)$ denotes
 - $N(x)$ if $B = N$,
 - $P(x, _)$ if $B = \exists P$,
 - $P(_, x)$ if $B = \exists P^-$,
 - $U(x, _)$ if $B = \delta(U)$,
 where ‘ $_$ ’ stands for an anonymous existentially quantified variable,
- An assertion $R(x, y)$ denotes
 - $P(x, y)$ if $R = P$,
 - $P(y, x)$ if $R = P^-$,
- An assertion $Z(x, y)$ denotes
 - $U(x, y)$ if $Z = U$,
 - $P(x, y)$ if $Z = P$,
 - $P(y, x)$ if $Z = P^-$,

■

Q-UNSAT-FOL
Query Abbreviation

Definition 2.42 (Q-UNSAT-FOL Query Abbreviation). We define several abbreviations for FOL queries, which make use the abbreviations in Definition 2.41, as follows:

$$\begin{aligned}
 q_{\text{unsat}}^f((\text{funct } Z), x, y, z) &= Z(x, y) \wedge Z(x, z) \wedge y \neq z; \\
 q_{\text{unsat}}^n(B_1 \sqsubseteq \neg B_2, x) &= B_1(x) \wedge B_2(x); \\
 q_{\text{unsat}}^n(R_1 \sqsubseteq \neg R_2, x, y) &= R_1(x, y) \wedge R_2(x, y); \\
 q_{\text{unsat}}^n(U_1 \sqsubseteq \neg U_2, x, y) &= U_1(x, y) \wedge U_2(x, y)
 \end{aligned}$$

■

Notice that the queries above can also be used to check whether the corresponding TBox assertion is violated or not. Next, in the following we define a boolean query Q_{unsatFOL}^T that can be used to check the consistency of a KB.

Q-UNSAT-FOL

Definition 2.43 (Q-UNSAT-FOL). Given a $DL\text{-}Lite_A$ TBox T , a query Q_{unsatFOL}^T is a boolean FOL query of the following form:

$$\begin{aligned}
 Q_{\text{unsatFOL}}^T = & \bigvee_{T \models (\text{funct } Z)} \exists x, y, z. q_{\text{unsat}}^f((\text{funct } Z), x, y, z) \vee \\
 & \bigvee_{T \models B_1 \sqsubseteq \neg B_2} \exists x. q_{\text{unsat}}^n(B_1 \sqsubseteq \neg B_2, x) \vee \\
 & \bigvee_{T \models R_1 \sqsubseteq \neg R_2} \exists x, y. q_{\text{unsat}}^n(R_1 \sqsubseteq \neg R_2, x, y) \vee \\
 & \bigvee_{T \models U_1 \sqsubseteq \neg U_2} \exists x, y. q_{\text{unsat}}^n(U_1 \sqsubseteq \neg U_2, x, y)
 \end{aligned}$$

■

Later on, when we do not want to distinguish between values and objects (thus we drop attributes), we drop the last disjunction of the query above.

Theorem 2.44 (FO Rewritability of Satisfiability Check in $DL\text{-}Lite_A$ [50]). *Given a KB $\langle T, A \rangle$, we have $\text{ANS}(Q_{\text{unsatFOL}}^T, A) = \text{true}$ (i.e., $A \models Q_{\text{unsatFOL}}^T$) if and only if A is T -inconsistent.* \square

2.6 History Preserving μ -Calculus

We now shift to reasoning about processes (the dynamic aspect). As a start, we briefly explain a temporal logic named History preserving μ -calculus. History preserving μ -calculus is a first order variant of μ -calculus [170, 149], one of the most powerful temporal logics, which subsumes LTL, PSL, and CTL* [83]. It was originally introduced in [24, 121]. Precisely, the work in [24] proposes a history preserving μ -calculus called $\mu\mathcal{L}_A$ and the work of [121] proposes a history preserving μ -calculus called $\mu\mathcal{L}_A^{\text{EQL}}$ (Note that the work in [121] originally call such logic also $\mu\mathcal{L}_A$, we use the name $\mu\mathcal{L}_A^{\text{EQL}}$ in order to differentiate it with the one in [24]). The different between $\mu\mathcal{L}_A$ and $\mu\mathcal{L}_A^{\text{EQL}}$ formulas is in the atomic parts of the formulas. The former consider DI-FOL queries as the atomic components of the formulas while the latter consider DI-ECQ queries.

2.6.1 History Preserving μ -Calculus with ECQ-Query ($\mu\mathcal{L}_A^{\text{EQL}}$)

The logic $\mu\mathcal{L}_A^{\text{EQL}}$ combines the standard temporal operators of μ -calculus with DI-ECQ queries over the states. First order quantification is interpreted with an active domain semantics, i.e., it ranges over those constants that are explicitly present in the current ABox, and fully interacts with temporal modalities, i.e., it applies *across* states. The $\mu\mathcal{L}_A^{\text{EQL}}$ syntax is:

$$\Phi := Q \mid \neg\Phi \mid \Phi_1 \vee \Phi_2 \mid \exists x.\Phi \mid \langle \neg \rangle \Phi \mid Z \mid \mu Z.\Phi \quad \text{Syntax of } \mu\mathcal{L}_A^{\text{EQL}}$$

where Q is a possibly open DI-ECQ query, Z is a second-order variable denoting a predicate (of arity 0), and μ is the least fixpoint operator, parametrized with the free variables of its bounding formula. Additionally, the following standard abbreviations hold:

- $\forall x.\Phi = \neg(\exists x.\neg\Phi)$,
- $\Phi_1 \wedge \Phi_2 = \neg(\neg\Phi_1 \vee \neg\Phi_2)$,
- $[\neg]\Phi = \neg\langle \neg \rangle \neg\Phi$, and
- $\nu Z.\Phi = \neg\mu Z.\neg\Phi[Z/\neg Z]$.

Given a KB $\langle T, A_0 \rangle$, we call a $\mu\mathcal{L}_A^{\text{EQL}}$ formula Φ *is over* $\langle T, A_0 \rangle$ if each query Q in Φ is a DI-ECQ query over $\langle T, A_0 \rangle$ (i.e., each atom in Q has either a concept, role or attribute name in $\text{VOC}(T)$ as its predicate, and Q might uses constants in $\text{ADOM}(A_0)$).

The semantics of $\mu\mathcal{L}_A^{\text{EQL}}$ formulae is defined over KB transition systems defined as follows:

Definition 2.45 (KB Transition System). A *KB transition system* \mathcal{T} is a tuple $\langle \Delta, T, \Sigma, s_0, \text{abox}, \Rightarrow \rangle$, where

- Δ is a countably infinite set of constants;

KB Transition System

- T is a *DL-Lite_A* TBox;
- Σ is a (possibly infinite) set of states;
- $s_0 \in \Sigma$ is the initial state;
- $abox$ is a function that, given a state $s \in \Sigma$, returns an ABox associated to s ;
- $\Rightarrow \subseteq \Sigma \times \Sigma$ is a transition relation between pairs of states.

■

Given a transition system \mathcal{T} , in order to assign the meaning to $\mu\mathcal{L}_A^{\text{EQL}}$ formulas, the following notions are introduced:

- A *individual variable valuation* v , i.e., a mapping from individual variables x to Δ .
- A *predicate variable valuation* V , i.e., a mapping from the predicate variables Z to a subset of Σ .

As for notations, since the individual variable valuation and the predicate variable valuation are substitutions, here we also use the notation that is defined in Section 2.2 as well (e.g., we write $v[x/c]$ to denote a valuation obtained from v such that $v[x/c](x) = c$, and $v[x/c](y) = v(y)$ if $y \neq x$, etc).

Semantics of $\mu\mathcal{L}_A^{\text{EQL}}$

The meaning of $\mu\mathcal{L}_A^{\text{EQL}}$ formulas are assigned by associating to \mathcal{T} , v and V an *extension function* $(\cdot)_{v,V}^{\mathcal{T}}$, which maps $\mu\mathcal{L}_A^{\text{EQL}}$ formulas to subsets of Σ . The extension function $(\cdot)_{v,V}^{\mathcal{T}}$ is defined inductively as follows:

$$\begin{aligned}
(Q)_{v,V}^{\mathcal{T}} &= \{s \in \Sigma \mid \text{CERT}(Qv, T, abox(s)) = \text{true}\} \\
(\exists x.\Phi)_{v,V}^{\mathcal{T}} &= \{s \in \Sigma \mid \exists d.d \in \text{ADOM}(abox(s)) \text{ and } s \in (\Phi)_{v[x/d],V}^{\mathcal{T}}\} \\
(Z)_{v,V}^{\mathcal{T}} &= V(Z) \subseteq \Sigma \\
(\neg\Phi)_{v,V}^{\mathcal{T}} &= \Sigma - (\Phi)_{v,V}^{\mathcal{T}} \\
(\Phi_1 \vee \Phi_2)_{v,V}^{\mathcal{T}} &= (\Phi_1)_{v,V}^{\mathcal{T}} \cup (\Phi_2)_{v,V}^{\mathcal{T}} \\
(\langle \rightarrow \rangle \Phi)_{v,V}^{\mathcal{T}} &= \{s \in \Sigma \mid \exists s'. s \Rightarrow s' \text{ and } s' \in (\Phi)_{v,V}^{\mathcal{T}}\} \\
(\mu Z.\Phi)_{v,V}^{\mathcal{T}} &= \bigcap \{\mathcal{E} \subseteq \Sigma \mid (\Phi)_{v,V[Z/\mathcal{E}]}^{\mathcal{T}} \subseteq \mathcal{E}\}
\end{aligned}$$

When Φ is a closed formula, $(\Phi)_{v,V}^{\mathcal{T}}$ does not depend on v or V , and we denote the extension of Φ simply by $(\Phi)^{\mathcal{T}}$. A closed formula Φ holds in a state $s \in \Sigma$ if $s \in (\Phi)^{\mathcal{T}}$. In this case, we write $\mathcal{T}, s \models \Phi$. A closed formula Φ holds in \mathcal{T} , briefly \mathcal{T} *satisfies* Φ , if $\mathcal{T}, s_0 \models \Phi$ (In this situation we write $\mathcal{T} \models \Phi$).

2.6.2 History Preserving μ -Calculus with FOL-Query ($\mu\mathcal{L}_A$)

The logic $\mu\mathcal{L}_A$ is similar to $\mu\mathcal{L}_A^{\text{EQL}}$ except that the atomic formulas are DI-FOL queries instead of DI-ECQ queries. Moreover, first order quantification is interpreted with an active domain semantics, i.e., it ranges over those constants that are explicitly present in the current database instance. Formally the syntax of $\mu\mathcal{L}_A$ is as follows:

Syntax of $\mu\mathcal{L}_A$

$$\Phi := Q \mid \neg\Phi \mid \Phi_1 \vee \Phi_2 \mid \exists x.\Phi \mid \langle \rightarrow \rangle \Phi \mid Z \mid \mu Z.\Phi$$

where Q is a possibly open DI-FOL query, and the others are the same as in $\mu\mathcal{L}_A^{\text{EQL}}$.

Given a database instance \mathbf{I} which conforms to a database schema \mathcal{R} , we call a $\mu\mathcal{L}_A$ formula Φ *is over* \mathcal{R} and \mathbf{I} if each query Q in Φ is a DI-FOL query over \mathcal{R} and

\mathbf{I} (i.e., the predicates of each atom in Q is a relation schema in \mathcal{R} , and Q might uses constants in $\text{ADOM}(\mathbf{I})$).

The semantics of $\mu\mathcal{L}_A$ formulae is defined over database transition systems defined as follows:

Definition 2.46 (Database Transition System). A *database transition system* \mathcal{T} is a tuple $\langle \Delta, \mathcal{R}, \Sigma, s_0, db, \Rightarrow \rangle$, where: Database Transition System

- Δ is a countably infinite set of constants;
- \mathcal{R} is a database schema;
- Σ is a (possibly infinite) set of states;
- $s_0 \in \Sigma$ is the initial state;
- db is a function that, given a state $s \in \Sigma$, returns the database associated to s , which is made up of constants in Δ and conforms to \mathcal{R} ;
- $\Rightarrow \subseteq \Sigma \times \Sigma$ is a transition relation between pairs of states.

■

The definition of the extension function $(\cdot)_{v,V}^{\mathcal{T}}$, which is used to assign meaning to $\mu\mathcal{L}_A$ formula, is the same as in $\mu\mathcal{L}_A^{\text{EQL}}$ except for the following Semantics of $\mu\mathcal{L}_A$

$$\begin{aligned} (Q)_{v,V}^{\mathcal{T}} &= \{s \in \Sigma \mid \text{ANS}(Qv, db(s)) = \text{true}\} \\ (\exists x. \Phi)_{v,V}^{\mathcal{T}} &= \{s \in \Sigma \mid \exists d. d \in \text{ADOM}(db(s)) \text{ and } s \in (\Phi)_{v[x/d],V}^{\mathcal{T}}\} \end{aligned}$$

The other notions that is defined in $\mu\mathcal{L}_A^{\text{EQL}}$ (e.g., \mathcal{T} satisfies Φ) is defined similarly as in $\mu\mathcal{L}_A$.

2.7 Data Centric Dynamic Systems (DCDSs)

Data Centric Dynamic Systems (DCDSs) [24, 23] provide an abstract model and formal foundation for various artifact-centric systems [147, 125, 85]. They capture the essence of systems in which both data and processes are first class citizens, and thus they provide a holistic view of the system. Furthermore, a DCDS also captures the manipulation of the data that is done by the available processes in the system. Here, the set Δ of constants denotes all possible values in the system. Additionally, we consider a finite set of distinguished constants $\Delta_0 \subset \Delta$, and we also make use of a finite set \mathcal{F} of *function symbols* that represent service calls, and can be used to inject fresh values (constants) into the system.

2.7.1 DCDSs Formalism

Technically, a DCDS consists of: (1) the *data component* which represents the data of interest in the application, and (2) the *process component* which represents the progression mechanism for the DCDS. In order to formally define the data component, we first introduce several preliminaries as follows:

Definition 2.47 (Equality Constraint (EC)). Given a database instance \mathbf{I} which conforms to a database schema \mathcal{R} , an *equality constraint (EC)* E over \mathcal{R} and \mathbf{I} is an expression of the form Equality Constraint (EC)

$$Q(\vec{x}) \rightarrow \bigwedge_{i=1,\dots,n} x_i = y_i,$$

where $Q(\vec{x})$ is a DI-FOL query over \mathcal{R} and \mathbf{I} , and x_i and y_i are either a variable in \vec{x} or a constant in $\text{ADOM}(\mathbf{I})$. ■

EC Satisfaction **Definition 2.48** (EC Satisfaction). Given a database instance \mathbf{I} and an equality constraint $E = Q(\vec{x}) \rightarrow \bigwedge_{i=1,\dots,n} x_i = y_i$, we say \mathbf{I} *satisfies* E if for each substitution $\sigma \in \text{ANS}(Q, \mathbf{I})$, it holds that $x_i\sigma = y_i\sigma$. ■

Given a database instance \mathbf{I} which conforms to a database schema \mathcal{R} , and a set \mathcal{E} of equality constraints over \mathcal{R} , we say \mathbf{I} *satisfies* \mathcal{E} if \mathbf{I} satisfies each $E \in \mathcal{E}$. Intuitively, given a database instance \mathbf{I} which conforms to a database schema \mathcal{R} , the equality constraints over \mathbf{I} and \mathcal{R} can be used to express some constraints to restrict the relation instances (facts) that can be contained in \mathbf{I} .

Now we define the data component of DCDSs as a tuple of a relational schema, a finite set of constraints, and an initial database instance which conforms to the given relational schema. Formally, it is stated below:

DCDS Data Component **Definition 2.49** (DCDS Data Component). A *DCDS data component* is a tuple $D = \langle \mathcal{R}, \mathbf{I}_0, \mathcal{E} \rangle$ where:

- \mathcal{R} is a *database schema*,
- \mathbf{I}_0 is an *initial database instance* which conforms to the schema \mathcal{R} . Intuitively, it represents the initial data of the system.
- \mathcal{E} is a finite set of *equality constraints* over \mathcal{R} and \mathbf{I}_0 .

Additionally, we impose that \mathbf{I}_0 satisfies each equality constraint $E \in \mathcal{E}$. ■

As uttered before, the process component constitutes the mechanism to evolve (the data in) the system. Basically, it consists of:

1. *Actions* which change the data from one state to another state, and might also issue *service calls* that introduce new values (constants) to the system during their execution (i.e., representing the interaction between the system and external user/environment);
2. *Condition-action rules* which specify when and with which parameters a certain action can be executed.

Formally, actions and condition-action rules is defined as follows.

DCDS Action **Definition 2.50** (DCDS Action). Given a data component $D = \langle \mathcal{R}, \mathbf{I}_0, \mathcal{E} \rangle$, a *DCDS action* α over \mathcal{R} and \mathbf{I}_0 is an expression of the form

$$\alpha(p_1, \dots, p_n) : \{e_1, \dots, e_m\},$$

where:

- α is an *action name*,
- the sequence p_1, \dots, p_n of variables are *action parameters*,
- $\{e_1, \dots, e_m\}$ is a set of *DCDS action effects* (briefly effects). Each effect e_i has the form

$$q^+ \wedge Q^- \rightsquigarrow E,$$

where:

- $q^+ \wedge Q^-$ is a DI-FOL query over \mathcal{R} and $\text{ADOM}(\mathbf{I}_0)$, that might also includes action parameters \vec{p} as its terms and uses constants in Δ_0 . Additionally, the query q^+ is a UCQ, and the query Q^- is an FOL query whose free variables are included in those of q^+ .

- E is a set of atoms whose predicates are relation schemas in \mathcal{R} , which includes as terms:
 - * constants in Δ_0 ,
 - * action parameters,
 - * free variables of q^+ , and
 - * skolem terms $f(\vec{t})$ (representing a service call) formed by applying a function $f \in \mathcal{F}$ to either constants in Δ_0 , action parameters, or free variables of q^+ .

For brevity, depending on the situation, we sometimes also say that α is a DCDS action over D . ■

As for notation, given an action $\alpha(p_1, \dots, p_n) : \{e_1, \dots, e_m\}$, we write $\text{EFF}(\alpha)$ to denote the set $\{e_1, \dots, e_m\}$ of effects of α . Later on, when it is clear from the context, a DCDS action is simply called action (e.g., in this section we only consider DCDS action, thus it is clear that when we say an action, it means that we refer to a DCDS action). Similarly, for DCDS action effects, for compactness, when it is clear, we simply call them action effects.

The intuition of the actions definition above is as follows:

1. Before an action $\alpha(p_1, \dots, p_n) : \{e_1, \dots, e_m\}$ can be executed, the action parameters p_1, \dots, p_n need to be instantiated with constants from Δ .
2. Intuitively, in an effect of the form $q^+ \wedge Q^- \rightsquigarrow E$, the query q^+ selects the tuples to instantiate the atoms in E , and Q^- filters away some of them.
3. The skolem terms (that might appear) in E represent service calls, and during an action execution, they will be substituted with constants from Δ (representing the results of service call).

Definition 2.51 (DCDS Condition-Action Rule). Given a data component $D = \langle \mathcal{R}, \mathbf{I}_0, \mathcal{E} \rangle$, and a set \mathcal{A} of DCDS actions over D , a *DCDS condition-action rule over \mathcal{R} , \mathbf{I}_0 , and \mathcal{A}* is an expression of the form

*DCDS
Condition-Action
Rule*

$$Q(p_1, \dots, p_n) \mapsto \alpha(p_1, \dots, p_n),$$

where

- α is an action $\alpha \in \mathcal{A}$, and
- Q is a DI-FOL query over \mathcal{R} and \mathbf{I}_0 whose free variables are exactly the parameters of α . Additionally, Q might also uses constants in Δ_0 .

For brevity, we often also say that the above condition-action rule is a DCDS condition-action rule over D and \mathcal{A} . ■

Later on, when it is clear from the context, a DCDS condition-action rule is simply called condition-action rule. Intuitively, the query Q in the left hand side of $Q(p_1, \dots, p_n) \mapsto \alpha(p_1, \dots, p_n)$ expresses a condition which determines when the action α can be executed. More precisely, the action α is executable if the query Q is successfully evaluated. Furthermore, the query Q is also used to obtain the values (constants) to instantiate the parameters of α (i.e., the answers to query Q is also used to instantiate the parameters of α).

Now, we define the process component of DCDSs composed by a set of actions and a set of condition-action rules.

DCDS Process
Component

Definition 2.52 (DCDS Process Component). Given a data component $D = \langle \mathcal{R}, \mathbf{I}_0, \mathcal{E} \rangle$, a *DCDS process component* over D is a tuple $P = \langle \mathcal{A}, \varrho \rangle$ where:

- \mathcal{A} is a finite set of actions over D ,
- ϱ is a finite set of condition-action rules over D and \mathcal{A} that form the specification of the *DCDS process* (which tells at any moment which actions can be executed).

■

Having the necessary ingredients in hand (i.e., data and process component), we are now ready to formally define a DCDS as follows:

Data Centric
Dynamic System
(DCDS)

Definition 2.53 (Data Centric Dynamic System (DCDS)). A Data Centric Dynamic System (DCDS) $\mathcal{S} = \langle D, P \rangle$, where

- $D = \langle \mathcal{R}, \mathbf{I}_0, \mathcal{E} \rangle$ is a DCDS data component and
- $P = \langle \mathcal{A}, \varrho \rangle$ is a DCDS process component over D .

Additionally, we assume that $\text{ADOM}(\mathbf{I}_0) \subseteq \Delta_0$.

■

Example 2.54. Recall our running example scenario in Section 2.1. Here we will model such order processing scenario in a DCDS. In addition to this scenario, in the DCDS that we specify below, we do not strictly enforce that the order of the processing flow must be followed sequentially. For instance, the operation of approving an order might be followed by another order approval operation. However, for each specific order, we enforce that it is processed sequentially according to the order processing flow described in Section 2.1 (for example, before the company delivers an order, the company must perform a quality check operation). We now specify a DCDS $\mathcal{S} = \langle D, P \rangle$ where the data component $D = \langle \mathcal{R}, \mathbf{I}_0, \mathcal{E} \rangle$ and the process component $P = \langle \mathcal{A}, \varrho \rangle$ of \mathcal{S} are specified below.

As for the database schema, we utilize the database schema \mathcal{R} that is specified in Example 2.6. The initial database is specified as follows

$$\mathbf{I}_0 = \{\text{ORDER}(123, \text{chair}, \text{received}, 456, \text{NULL}, \text{NULL}, \text{NULL}, \text{NULL}, \text{ecodesign}), \\ \text{ORDER}(321, \text{table}, \text{approved}, 654, \text{NULL}, \text{NULL}, \text{NULL}, \text{NULL}, \text{NULL})\},$$

and we consider empty equality constraint $\mathcal{E} = \{\}$.

For the process component, we specify the set \mathcal{A} of actions containing the following actions:

1. `approveOrder(x)`, which intuitively changes the status of an order with id x into “approved”. Formally it is specified as follows:

$$\begin{aligned} \text{approveOrder}(x) : \{ \\ & \exists x_3. \text{ORDER}(x_1, \dots, x_9) \wedge x_1 = x \rightsquigarrow \{ \\ & \quad \text{ORDER}(x_1, x_2, \text{"approved"}, x_4, x_5, \\ & \quad \quad \text{NULL}, \text{NULL}, x_8, x_9) \\ & \quad \}, \\ & \text{ORDER}(x_1, \dots, x_9) \wedge x_1 \neq x \rightsquigarrow \{\text{ORDER}(x_1, \dots, x_9)\}, \\ & \text{DELIVERED_ORDER}(x_1, x_2) \rightsquigarrow \{\text{DELIVERED_ORDER}(x_1, x_2)\} \\ & \} \end{aligned}$$

Technically, the action above changes the status of the order with id x into "approved" while keeping the other database entries stay the same.

2. `prepareOrders()`, which intuitively prepares several things that are needed for further processing steps of each approved order. For each approved order, this action prepares the design of the corresponding order by calling an external service `GETDESIGN(x)`. Moreover, this action also retrieves the information about the corresponding designer by calling a service `GETDESIGNER(x)`, and assign the assembling location for the corresponding order by calling a service `ASSIGNASSEMBLINGLOC(x)`. Formally it is specified as follows:

$$\begin{aligned} \text{prepareOrders}() : \{ \\ & \exists x_5 x_8 x_9. \text{ORDER}(x_1, x_2, \text{"approved"}, x_4, \dots, x_9) \rightsquigarrow \{ \\ & \quad \text{ORDER}(x_1, x_2, \text{"approved"}, x_4, \text{GETDESIGNER}(x), x_6, x_7, \\ & \quad \quad \text{ASSIGNASSEMBLINGLOC}(x), \text{GETDESIGN}(x)) \\ & \quad \}, \\ & \text{ORDER}(x_1, \dots, x_9) \wedge x_3 \neq \text{"approved"} \rightsquigarrow \{\text{ORDER}(x_1, \dots, x_9)\}, \\ & \text{DELIVERED_ORDER}(x_1, x_2) \rightsquigarrow \{\text{DELIVERED_ORDER}(x_1, x_2)\} \\ & \} \end{aligned}$$

3. `assembleOrders()`, which represents the step of assembling several components into the corresponding ordered furniture. This action also acquires the information about the assembler and the assembling location by calling external service calls `GETASSEMBLER(x)` and `GETASSEMBLINGLOC(x)` respectively. Additionally, this action only assembles every approved order that already has a design. Formally this action is specified as follows:

$$\begin{aligned} \text{assembleOrders}() : \{ \\ & \exists x_6 x_8. \text{ORDER}(x_1, x_2, \text{"approved"}, x_4, x_5, x_6, x_7, x_8, x_9) \wedge x_9 \neq \text{NULL} \rightsquigarrow \{ \\ & \quad \text{ORDER}(x_1, x_2, \text{"assembled"}, x_4, x_5, \text{GETASSEMBLER}(x), \\ & \quad \quad x_7, \text{GETASSEMBLINGLOC}(x), x_9) \\ & \quad \}, \\ & \text{ORDER}(x_1, \dots, x_9) \wedge x_3 \neq \text{"approved"} \rightsquigarrow \{\text{ORDER}(x_1, \dots, x_9)\} \\ & \text{ORDER}(x_1, x_2, \text{"approved"}, x_4, \dots, x_9) \wedge x_9 = \text{NULL} \rightsquigarrow \\ & \quad \{\text{ORDER}(x_1, x_2, \text{"approved"}, x_4, \dots, x_9)\}, \\ & \text{DELIVERED_ORDER}(x_1, x_2) \rightsquigarrow \{\text{DELIVERED_ORDER}(x_1, x_2)\} \\ & \} \end{aligned}$$

4. `checkAssembledOrders()`, which models the quality check process for each assembled order. This action calls an external service `GETQUALITYCONTROLLER(x)`

in order to obtain the quality controller officer who performs the task. Formally it is specified as follows:

$$\begin{aligned} \text{checkAssembledOrders}() : \{ \\ & \text{ORDER}(x_1, x_2, \text{"assembled"}, x_4, x_5, x_6, \text{NULL}, x_8, x_9) \rightsquigarrow \{ \\ & \quad \text{ORDER}(x_1, x_2, \text{"assembled"}, x_4, x_5, x_6, \\ & \quad \quad \text{GETQUALITYCONTROLLER}(x), x_8, x_9) \} \\ & \text{ORDER}(x_1, x_2, \text{"assembled"}, x_4, x_5, x_6, x_7, x_8, x_9) \wedge x_7 \neq \text{NULL} \rightsquigarrow \{ \\ & \quad \text{ORDER}(x_1, x_2, \text{"assembled"}, x_4, x_5, x_6, x_7, x_8, x_9) \} \\ & \text{ORDER}(x_1, \dots, x_9) \wedge x_3 \neq \text{"assembled"} \rightsquigarrow \{ \text{ORDER}(x_1, \dots, x_9) \}, \\ & \text{DELIVERED_ORDER}(x_1, x_2) \rightsquigarrow \{ \text{DELIVERED_ORDER}(x_1, x_2) \} \\ & \} \end{aligned}$$

5. `deliverOrders()`, which delivers each assembled order that has passed the quality control process. Formally this action is specified as follows:

$$\begin{aligned} \text{deliverOrders}() : \{ \\ & \text{ORDER}(x_1, x_2, \text{"assembled"}, x_4, x_5, x_6, \text{NULL}, x_8, x_9) \rightsquigarrow \{ \\ & \quad \text{ORDER}(x_1, x_2, \text{"assembled"}, x_4, x_5, x_6, \text{NULL}, x_8, x_9) \\ & \quad \}, \\ & \text{ORDER}(x_1, x_2, \text{"assembled"}, x_4, x_5, x_6, x_7, x_8, x_9) \wedge x_7 \neq \text{NULL} \rightsquigarrow \{ \\ & \quad \text{ORDER}(x_1, x_2, \text{"delivered"}, x_4, x_5, x_6, x_7, x_8, x_9), \\ & \quad \text{DELIVERED_ORDER}(x_1, \text{GETDELIVERYDATE}(x_1)), \\ & \quad \}, \\ & \text{ORDER}(x_1, \dots, x_9) \wedge x_3 \neq \text{"assembled"} \rightsquigarrow \{ \text{ORDER}(x_1, \dots, x_9) \}, \\ & \text{DELIVERED_ORDER}(x_1, x_2) \rightsquigarrow \{ \text{DELIVERED_ORDER}(x_1, x_2) \} \\ & \} \end{aligned}$$

Furthermore, the set ϱ of condition action rules is specified as follows:

- $\exists x_2 x_4 x_5 x_6 x_7 x_8 x_9. \text{ORDER}(x_1, x_2, \text{"received"}, x_4, \dots, x_9) \mapsto \text{approveOrder}(x_1),$
- $\exists x_1 x_2 x_4 x_5 x_6 x_7 x_8 x_9. \text{ORDER}(x_1, x_2, \text{"approved"}, x_4, \dots, x_9) \mapsto \text{prepareOrders}(),$
- $\exists x_1 x_2 x_4 x_5 x_6 x_7 x_8 x_9. \text{ORDER}(x_1, x_2, \text{"approved"}, x_4, \dots, x_9) \wedge x_9 \neq \text{NULL} \\ \mapsto \text{assembleOrders}(),$
- $\exists x_1 x_2 x_4 x_5 x_6 x_8 x_9. \text{ORDER}(x_1, x_2, \text{"assembled"}, x_4, x_5, x_6, \text{NULL}, x_8, x_9) \\ \mapsto \text{checkAssembledOrders}(),$
- $\exists x_1 x_2 x_4 x_5 x_6 x_7 x_8 x_9. \text{ORDER}(x_1, x_2, \text{"assembled"}, x_4, x_5, x_6, x_7, x_8, x_9) \\ \wedge x_7 \neq \text{NULL} \mapsto \text{deliverOrders}(),$

The intuition for each condition action rules above is consecutively presented below:

- the first rule states that if there exists an order with the status “received” and has an ID x_1 , then we can fire the execution of action `approveOrder/1` with the argument x_1 (i.e., `approveOrder(x_1)`).

- The second rule says that we can execute the action `prepareOrders/0` in case there exists at least one approved order.
- Next, the third rule encodes the condition where the execution of action `assembleOrders/0` can be fired in case there exists an approved order that already has a design.
- The fourth rule indicates that if there exists an assembled order that has not been checked, we can execute the action `checkAssembledOrders/0`
- Finally, the last rule specifies that whenever there exists an assembled order that has been checked, we can execute the action `deliverOrders/0`.

2.7.2 DCDSs Execution Semantics

The semantic of DCDS is defined in terms of a possibly infinite transition system whose states are labeled by databases and where transitions represent the execution of actions. Such transition system represents all possible computations that the process component can do on the data component starting from the initial database instance (i.e., all possible manipulations of data by actions).

During the execution, an action can issue service calls. In [24], there are two kinds of service calls semantics that are considered, namely *deterministic* and *non-deterministic* service calls semantics. In the deterministic service calls semantics, along a run of the system, whenever a service call is issued with the same input parameters, it will return the same value (constant). On the other hand, in the non-deterministic service calls semantics, along a run of the system, two different issues of a service call with the same input parameters might return distinct results. Here, in this thesis, we assume that the semantics of service calls is deterministic.

To enforce the deterministic service calls semantics, the transition systems of a DCDS remembers the results of previous service calls in a so-called service call map defined as follows.

Definition 2.55 (Service Call Map). A *service call map* is a partial function

Service Call Map

$$m : \mathbf{SC} \rightarrow \Delta,$$

where \mathbf{SC} is the set $\{f(v_1, \dots, v_n) \mid f/n \in \mathcal{F} \text{ and } \{v_1, \dots, v_n\} \subseteq \Delta\}$ of (skolem terms representing) *service calls*. ■

Technically, to provide the semantics of DCDSs, we consider database transition systems (as defined in Definition 2.46), i.e., transition systems of the form $\langle \Delta, \mathcal{R}, \Sigma, s_0, db, \Rightarrow \rangle$, where:

- \mathcal{R} is a database schema;
- Σ is a set of states;
- $s_0 \in \Sigma$ is the initial state;
- db is a function that, given a state $s \in \Sigma$, returns the database associated to s , which is made up of constants in Δ and conforms to \mathcal{R} ;
- $\Rightarrow \subseteq \Sigma \times \Sigma$ is a transition relation between pairs of states.

In addition, to realize DCDSs with deterministic service calls semantics, each state $s \in \Sigma$ of the transition system is defined as a tuple $\langle \mathbf{I}, m \rangle$, where \mathbf{I} is a database

instance and m is a service call map. As for notations related to the service call map, we use similar notations that is defined for substitutions in Section 2.2 (e.g., we write $f(c)/v \in m$ to say that a service call map m maps $f(c)$ to v).

Action Execution
Semantics in DCDS

In order to define the semantics of an *action execution* in DCDS, below we define the notion of when an action can be executed, and how the results of an action execution is constructed.

Executability of an
Action in DCDS

Definition 2.56 (Executability of an Action in DCDS). Let $\mathcal{S} = \langle D, P \rangle$ be a DCDS where $P = \langle \mathcal{A}, \varrho \rangle$. Given a database instance \mathbf{I} , an action $\alpha \in \mathcal{A}$ of the form $\alpha(\vec{p}) : \{e_1, \dots, e_m\}$, and a *parameter substitution* σ which substitute the parameters \vec{p} with constants taken from Δ . We say that α is *executable in \mathbf{I} with a parameter substitution σ* , if there exists a condition-action rule $Q(\vec{p}) \mapsto \alpha(\vec{p}) \in \varrho$ such that $\sigma \in \text{ANS}(Q, \mathbf{I})$. ■

With a little abuse of the definition, we sometimes say that an action α is *executable in a state s with a parameter substitution σ* if $s = \langle \mathbf{I}, m \rangle$ and α is executable in \mathbf{I} with a parameter substitution σ . Additionally, given an action $\alpha(p_1, \dots, p_n) : \{e_1, \dots, e_m\}$, and a database instance \mathbf{I} , we say that σ is a *legal parameter assignment* for α in \mathbf{I} if α is executable in \mathbf{I} with a parameter substitution σ . Moreover, we write $\alpha\sigma$ to denote a *grounded action* that is obtained by applying a legal parameter assignment σ to each $e \in \text{EFF}(\alpha)$ (i.e., substituting each occurrence of p_i (for $i \in \{1, \dots, n\}$) in e with a constant in Δ based on the substitution σ).

Example 2.57. Continuing our running example in Example 2.54, we have that the action `approveOrder/1` is executable in the state $s_0 = \langle \mathbf{I}_0, \emptyset \rangle$ with a parameter substitution σ , where σ is a substitution that substitutes the action parameter of `approveOrder/1` to 123. This is the case because we have that the query in the left hand side of the condition-action rule

$$\exists x_2 x_4 x_5 x_6 x_7 x_8 x_9. \text{ORDER}(x_1, x_2, \text{"received"}, x_4, \dots, x_9) \mapsto \text{approveOrder}(x_1),$$

is successfully evaluated over \mathbf{I}_0 and give an answer 123 (i.e., $\sigma(x_1) = 123$). In this case, we have that σ is the legal parameter assignment for α in \mathbf{I}_0 .

The execution result of a grounded action $\alpha\sigma$ is captured by a function $\text{DO}(\mathbf{I}, \alpha\sigma)$ which is formally defined as follows:

Computation of
DCDS Action
Execution Result

Definition 2.58 (Computation of DCDS Action Execution Result). Let $\mathcal{S} = \langle D, P \rangle$ be a DCDS where $P = \langle \mathcal{A}, \varrho \rangle$. Given a database instance \mathbf{I} , an action $\alpha \in \mathcal{A}$, and a legal parameter assignment σ for α in \mathbf{I} . The *execution result of $\alpha\sigma$ in \mathbf{I}* is computed by function $\text{DO}_{\text{DCDS}}(\mathbf{I}, \alpha\sigma)$ as follows:

$$\text{DO}_{\text{DCDS}}(\mathbf{I}, \alpha\sigma) = \left(\bigcup_{e_i = q_i^+ \wedge Q_i^- \rightsquigarrow E \text{ in } \text{EFF}(\alpha)} \bigcup_{\rho \in \text{ANS}((\llbracket q_i^+ \rrbracket \wedge Q_i^-) \sigma, \mathbf{I})} E\sigma\rho \right)$$

■

Intuitively, the execution result of α is obtained by collecting all facts in $E\sigma\rho$ of each effect $q^+ \wedge Q^- \rightsquigarrow E$ in $\text{EFF}(\alpha)$, where the set $E\sigma\rho$ of facts of $q^+ \wedge Q^- \rightsquigarrow E$ is obtained by substituting each variable in each atom in E with a constant in Δ based on the answers of the query $q^+ \wedge Q^-$ (i.e., substitution ρ) and the legal parameter assignment σ .

Note that there might be some facts in $\text{DO}_{\text{DCDS}}(\mathbf{I}, \alpha\sigma)$ that contain (ground) skolem terms. Intuitively, it means that some service calls have to be issued in order to replace it with some values (constants) and get a proper database instance as the result of an action execution. We denote by $\text{CALLS}(\text{DO}_{\text{DCDS}}(\mathbf{I}, \alpha\sigma))$ the set of such ground service calls (ground skolem terms), and by $\text{EVAL}(\mathbf{I}, \alpha\sigma)$ the set of substitutions that replace such calls with concrete constants taken from Δ . Specifically, $\text{EVAL}(\mathbf{I}, \alpha\sigma)$ is defined as

Service Call
Evaluation

$$\text{EVAL}(\mathbf{I}, \alpha\sigma) = \{\theta \mid \theta \text{ is a total function } \theta : \text{CALLS}(\text{DO}(\mathbf{I}, \alpha\sigma)) \rightarrow \Delta\}.$$

As for notations related to the substitution $\theta \in \text{EVAL}(\mathbf{I}, \alpha\sigma)$, we use similar notations, that is defined in Section 2.2 (e.g., we write $f(c)/v \in \theta$ to say that θ maps $f(c)$ to v).

Having the semantics of an action execution in place, given a DCDS $\mathcal{S} = \langle D, P \rangle$, we employ $\text{DO}_{\text{DCDS}}()$ and $\text{EVAL}()$ to define a transition relation $\text{EXEC}_{\mathcal{S}}$ connecting two states through an action execution as follows.

Definition 2.59 (DCDS Transition Relation $\text{EXEC}_{\mathcal{S}}$). Let $\mathcal{S} = \langle D, P \rangle$ be a DCDS where $P = \langle \mathcal{A}, \varrho \rangle$. Given a state $s = \langle \mathbf{I}, m \rangle$, a state $s' = \langle \mathbf{I}', m' \rangle$, an action $\alpha \in \mathcal{A}$, and a substitution σ . We have $\langle \langle \mathbf{I}, m \rangle, \alpha\sigma, \langle \mathbf{I}', m' \rangle \rangle \in \text{EXEC}_{\mathcal{S}}$ if the following holds:

DCDS Transition
Relation

1. α is *executable* in s with legal parameter assignment σ ;
2. there exists $\theta \in \text{EVAL}(\mathbf{I}, \alpha\sigma)$ such that for each skolem term $f(c) \in \text{DOM}(m) \cap \text{DOM}(\theta)$, we have $f(c)/v \in m$ if and only if $f(c)/v \in \theta$ (i.e., θ and m “agree” on the common skolem terms in their domains, in order to realize the deterministic service call semantics);
3. $\mathbf{I}' = \text{DO}_{\text{DCDS}}(\mathbf{I}, \alpha\sigma)\theta$;
4. $m' = m \cup \theta$ (i.e., updating the history of issued service calls).

■

When $\langle \langle \mathbf{I}, m \rangle, \alpha\sigma, \langle \mathbf{I}', m' \rangle \rangle \in \text{EXEC}_{\mathcal{S}}$, we equivalently write

$$\langle \mathbf{I}, m \rangle \xrightarrow{\alpha\sigma, \mathcal{S}} \langle \mathbf{I}', m' \rangle.$$

for easiness of reading. When it is clear from the context, we also often omit \mathcal{S} and simply write $\langle \mathbf{I}, m \rangle \xrightarrow{\alpha\sigma} \langle \mathbf{I}', m' \rangle$.

The transition system $\mathcal{T}_{\mathcal{S}}$ of DCDS \mathcal{S} , which provide the execution semantics of \mathcal{S} , is then formally defined as follows:

Definition 2.60 (DCDSs Transition System). Given a DCDS $\mathcal{S} = \langle D, P \rangle$ with $D = \langle \mathcal{R}, \mathbf{I}_0, \mathcal{E} \rangle$, and $P = \langle \mathcal{A}, \varrho \rangle$, the transition system $\mathcal{T}_{\mathcal{S}}$ is defined as $\langle \Delta, \mathcal{R}, \Sigma, s_0, db, \Rightarrow \rangle$ where

DCDSs Transition
System

- $s_0 = \langle \mathbf{I}_0, \emptyset \rangle$, and
- Σ and \Rightarrow are defined by simultaneous induction as the smallest sets satisfying the following properties:
 1. $s_0 \in \Sigma$;

2. if $\langle \mathbf{I}, m \rangle \in \Sigma$, then for all actions $\alpha \in \mathcal{A}$, for all legal parameter assignments σ for α in \mathbf{I} and for all $\langle \mathbf{I}', m' \rangle$ such that
 - a) $\langle \mathbf{I}, m \rangle \xrightarrow{\alpha\sigma, \mathcal{S}} \langle \mathbf{I}', m' \rangle$, and
 - b) \mathbf{I}' satisfies \mathcal{E} ,
 we have $\langle \mathbf{I}', m' \rangle \in \Sigma$, and $\langle \mathbf{I}, m \rangle \Rightarrow \langle \mathbf{I}', m' \rangle$.

■

Roughly speaking, the transition system of a DCDS, which provides its execution semantics, is obtained by nondeterministically applying every executable action starting from the initial database with corresponding legal parameter assignments, and considering each possible value (constant) returned by applying the involved service calls. Additionally, we restrict that an action with certain parameters is executable if the database instance produced by its execution satisfies the given equality constraints. Otherwise the action is considered as non executable with the chosen parameters.

Example 2.61. Continuing our running example in Example 2.54, the construction of transition system $\mathcal{T}_{\mathcal{S}}$ of DCDS \mathcal{S} is started from the initial state $s_0 = \langle \mathbf{I}_0, \emptyset \rangle$ where

$$\mathbf{I}_0 = \{\text{ORDER}(123, \text{chair}, \text{received}, 456, \text{NULL}, \text{NULL}, \text{NULL}, \text{NULL}, \text{ecodesign}), \\ \text{ORDER}(321, \text{table}, \text{approved}, 654, \text{NULL}, \text{NULL}, \text{NULL}, \text{NULL}, \text{NULL})\}.$$

An example of a successor of state s_0 is a state $s_1 = \langle \mathbf{I}_1, m_1 \rangle$, where

$$\mathbf{I}_1 = \{\text{ORDER}(123, \text{chair}, \text{approved}, 456, \text{NULL}, \text{NULL}, \text{NULL}, \text{NULL}, \text{ecodesign}), \\ \text{ORDER}(321, \text{table}, \text{approved}, 654, \text{NULL}, \text{NULL}, \text{NULL}, \text{NULL}, \text{NULL})\}, \\ m_1 = \emptyset$$

and s_1 is obtained from the execution of action `approveOrder/1` with the argument 123. The action `approveOrder/1` is executable with argument 123 in the state s_0 since we have that the query in the left hand side of the condition-action rule

$$\exists x_2 x_4 x_5 x_6 x_7 x_8 x_9. \text{ORDER}(x_1, x_2, \text{"received"}, x_4, \dots, x_9) \mapsto \text{approveOrder}(x_1),$$

is successfully evaluated and give an answer 123. As the result of executing this action, we have that now the status of order 123 become approved.

Another example of a successor of state s_0 is a state $s_2 = \langle \mathbf{I}_2, m_2 \rangle$, where

$$\mathbf{I}_2 = \{\text{ORDER}(123, \text{chair}, \text{received}, 456, \text{NULL}, \text{NULL}, \text{NULL}, \text{NULL}, \text{ecodesign}), \\ \text{ORDER}(321, \text{table}, \text{approved}, 654, \text{bob}, \text{NULL}, \text{NULL}, \text{bolzano}, \text{classicdesign})\}, \\ m_2 = \{[\text{GETDESIGNER}(321)) \rightarrow \text{bob}], [\text{ASSIGNASSEMBLINGLOC}(321)) \rightarrow \text{bolzano}], \\ [\text{GETDESIGN}(321)) \rightarrow \text{classicdesign}]\},$$

and s_2 is obtained from the execution of action `prepareOrders/0`. The action `prepareOrders/0` is executable in the state s_0 since we have that the query in the left hand side of the condition-action rule

$$\exists x_1 x_2 x_4 x_5 x_6 x_7 x_8 x_9. \text{ORDER}(x_1, x_2, \text{"approved"}, x_4, \dots, x_9) \mapsto \text{prepareOrders}(),$$

is successfully evaluated.

2.7.3 Verification of DCDSs

The interesting reasoning task in DCDSs is to verify whether the transition system of a given DCDS satisfies temporal properties of interest, specified in some first-order temporal logic. To specify the temporal properties, the work in [24] uses the temporal logic history preserving μ -calculus ($\mu\mathcal{L}_A$) (see Section 2.6). The verification problem of $\mu\mathcal{L}_A$ properties over DCDSs is then formally stated as follows:

Definition 2.62 (Verification of a $\mu\mathcal{L}_A$ property over a DCDS). Given a DCDS $\mathcal{S} = \langle D, P \rangle$ (with $D = \langle \mathcal{R}, \mathbf{I}_0, \mathcal{E} \rangle$) and a closed $\mu\mathcal{L}_A$ formula Φ over \mathcal{R} and \mathbf{I}_0 , the verification of a $\mu\mathcal{L}_A$ property Φ over \mathcal{S} is a problem to check whether $\mathcal{T}_{\mathcal{S}} \models \Phi$. ■

Verification of a $\mu\mathcal{L}_A$
Property Over a
DCDS

We also say a DCDS \mathcal{S} satisfies a closed $\mu\mathcal{L}_A$ formula Φ if $\mathcal{T}_{\mathcal{S}} \models \Phi$. As studied in [24], it has been shown that in general the verification of $\mu\mathcal{L}_A$ over DCDS is undecidable. However, the work in [24] has identified some restrictions to get the decidability. Here we briefly explain a semantic restriction that was introduced in [24], namely *run-boundedness*.

Definition 2.63 (Run of a DCDS Transition System). Given a DCDS \mathcal{S} , a run of $\mathcal{T}_{\mathcal{S}}$ is a (possibly infinite) sequence $s_0 s_1 \dots$ of states of $\mathcal{T}_{\mathcal{S}}$ such that $s_i \Rightarrow s_{i+1}$, for all $i \geq 0$. ■

Run of a DCDS
Transition System

Definition 2.64 (Run-bounded DCDS). Given a DCDS \mathcal{S} , we say \mathcal{S} is run-bounded if there exists an integer bound b such that for every run $\pi = s_0 s_1 \dots$ of $\mathcal{T}_{\mathcal{S}}$, we have that $|\bigcup_{s \text{ state of } \pi} \text{ADOM}(db(s))| < b$. ■

Run-bounded DCDS

Intuitively, run-boundedness requires that every run in the transition system cumulatively encounters at most a bounded number of constants. Unboundedly many constants can still be present in the overall system, provided that they do not accumulate in the same run.

Theorem 2.65 (Verification of $\mu\mathcal{L}_A$ over run-bounded DCDS [24]). Verification of $\mu\mathcal{L}_A$ properties over run-bounded DCDS is decidable and can be reduced to finite-state model checking.

KNOWLEDGE AND ACTION BASES (KABs)

Knowledge and Action Bases (KABs) [22, 121] have been proposed as a unified framework to simultaneously account for the static and dynamic aspects of an application domain. Essentially, it provides a semantically rich representation of the information on the domain of interest in terms of a DL KB and a set of actions to change such information over time, possibly introducing new objects.

Here we consider the KABs that are obtained by combining the framework in [121] with the action specification formalism in [146]. Specifically, rather than following the original KABs [121], in which at each action execution the state is reconstructed from scratch, we adopt the action formalism in [146], in which one specifies only the facts to add and those to delete from the current state. Additionally, radically different from [121] where service calls are not evaluated, in this work we evaluate the service calls (in the sense that we substitute each service call with a concrete value) when constructing the transition system. This service call evaluation semantics is similar to the work on Data Centric Dynamic Systems in [24].

In the following, we use $DL\text{-}Lite_{\mathcal{A}}$ for expressing knowledge bases and we also do not distinguish between objects and values (thus we drop attributes). Furthermore, we make use of a countably infinite set Δ of constants, which intuitively denotes all possible values in the system. Additionally, we consider a finite set of distinguished constants $\Delta_0 \subset \Delta$, and a finite set \mathcal{F} of *function symbols* that represents *service calls*, which abstractly account for the injection of fresh values (constants) from Δ into the system.

3.1 KABs Formalism

In a nutshell, a KAB is composed by a $DL\text{-}Lite_{\mathcal{A}}$ KB (see Definition 2.13), and an action base (consisting of actions and process) which represents the progression mechanism for KAB. In order to formally define KABs, we first introduce the notion of KAB actions as well as KAB process (which is specified as a finite set of condition-action rules).

Syntactically, an action in a KAB is formalized in the following definitions:

Definition 3.1 (KAB Action). Given a KB $\langle T, A_0 \rangle$, a *KAB action* α over $\langle T, A_0 \rangle$ is an expression of the form

$$\alpha(\vec{p}) : \{e_1, \dots, e_m\},$$

where

- α is the *action name*,
- \vec{p} are the *action parameters*, and

- $\{e_1, \dots, e_m\}$ is the set of *KAB action effects* (briefly effects). Each effect e_i has the form

$$[q_i^+] \wedge Q_i^- \rightsquigarrow \mathbf{add} F_i^+, \mathbf{del} F_i^-$$

where:

- $[q_i^+] \wedge Q_i^-$ is a DI-ECQ query over $\langle T, A_0 \rangle$, that might also includes action parameters \vec{p} as its terms and uses constants from Δ_0 . Additionally, the query $[q_i^+]$ is a UCQ, and the query Q_i^- is an ECQ whose free variables are included in those of $[q_i^+]$.
- F_i^+ is a set of atoms whose predicates are either concept or role names in $\text{voc}(T)$ and each having as terms: either constants in Δ_0 , action parameters \vec{p} , free variables of $[q_i^+]$, or skolem terms $f(\vec{t})$ (representing *service calls*) formed by applying a function $f \in \mathcal{F}$ to either constants in Δ_0 , action parameters, or free variables of $[q_i^+]$.
- F_i^- is a set of atoms whose predicates are either concept or role names in $\text{voc}(T)$, and each having as terms: either constants in Δ_0 , action parameters \vec{p} , or free variables of $[q_i^+]$.

■

Given an action α , we write $\text{EFF}(\alpha)$ to denote the *set of effects in α* . For brevity, when it is clear from the context, we often only write “action” (resp. “effect”) instead of “KAB action” (resp. “KAB action effect”). Furthermore, we also often omit the “ $\mathbf{add} F^+$ ” part (resp., the “ $\mathbf{del} F^-$ ” part) if $F^+ = \emptyset$ (resp., if $F^- = \emptyset$).

Intuitively, an action α is executed by grounding its parameters, and then applying its effects in parallel. For each effect the form

$$[q^+] \wedge Q^- \rightsquigarrow \mathbf{add} F^+, \mathbf{del} F^-,$$

the query $[q^+]$ intuitively selects the values to instantiate the atoms in F^+ as well as F^- , and Q^- filters *away* some of those values¹. Moreover, the skolem terms that might be contained in F^+ represent service calls, and during an action execution, they will be substituted with values from Δ (representing the results of service calls). Intuitively, the service calls represent the interactions between the system and the external environment. After each atom in F^+ (resp. F^-) has been instantiated and all service calls has been issued (i.e., has been substituted with the results of service calls), it becomes a set of assertions to be added (resp. deleted) to (resp. from) the ABox. The update induced by α is produced by adding and removing those assertions (the assertions to be added/deleted) to/from the current ABox, giving higher priority to additions (i.e., if the same assertion is asserted to be added and deleted during the same action execution step, then the assertion is added). All of these intuitions of an action execution will be formalized later when we introduce the semantics of an action execution.

The instantiations of action parameters are determined by condition-action rules. Additionally, a condition-action rule also determines when a certain action can be executed. Formally, a condition-action rule is defined as follows:

¹ to convey this intuition, we use the “ $+$ ” and “ $-$ ” superscript

Definition 3.2 (KAB Condition-Action Rule). Given a KB $\langle T, A_0 \rangle$, and a set Γ of KAB actions over $\langle T, A_0 \rangle$, a *KAB condition-action rule* over $\langle T, A_0 \rangle$ and Γ is an expression of the form

$$Q(\vec{p}) \mapsto \alpha(\vec{p}),$$

where:

- $\alpha \in \Gamma$ is a KAB action, and
- $Q(\vec{p})$ is a DI-ECQ over $\langle T, A_0 \rangle$, whose free variables are exactly the parameters of α . Additionally, $Q(\vec{p})$ might use constants in Δ_0 .

KAB
Condition-Action
Rule

For brevity, we often simply write $Q \mapsto \alpha$ instead of $Q(\vec{p}) \mapsto \alpha(\vec{p})$. Intuitively, the query Q expresses a condition when the action α can be executed. Moreover, if Q is an open query, then the answers of Q are used to instantiate the parameters of α .

Having the required machinery in hand, we are ready to formally define KABs as follows.

Definition 3.3 (Knowledge and Action Base).

A KAB is a tuple $\mathcal{K} = \langle T, A_0, \Gamma, \Pi \rangle$ where:

- T together with A_0 form a satisfiable *DL-Lite_A* KB $\langle T, A_0 \rangle$, where
 - T is a *DL-Lite_A* TBox that captures the intensional aspects of the domain of interest;
 - A_0 is the *initial DL-Lite_A* ABox, describing the initial configuration of data;
- Γ is a finite set of *KAB actions* over $\langle T, A_0 \rangle$ that evolve the ABox;
- Π is a finite set of *KAB condition-action rules* over $\langle T, A_0 \rangle$ and Γ forming a *process* (which tells at any moment which actions can be executed, and with which parameters).

Knowledge and
Action Base (KAB)
Formalism

Additionally, we assume that $\text{ADOM}(A_0) \subseteq \Delta_0$.

Roughly speaking, T and A_0 together form the *knowledge base* while Γ and Π form the *action base* which evolves the knowledge base. We assume that $\text{ADOM}(A_0) \subseteq \Delta_0$. Intuitively, the KB maintains the information of interest. A_0 represents the initial state of the system and, differently from T , it evolves and incorporates new information from the external world by executing actions Γ , according to the sequencing established by process Π .

Without loss of generality, in a KAB, we assume that for each action $\alpha \in \Gamma$ there exists at most one condition-action rule $Q(\vec{p}) \mapsto \alpha(\vec{p}) \in \Pi$. Notice that several condition-action rules $\{Q_1(\vec{p}) \mapsto \alpha(\vec{p}), \dots, Q_n(\vec{p}) \mapsto \alpha(\vec{p})\} \subseteq \Pi$ for an action $\alpha \in \Gamma$, can be compactly represented as a single condition-action rule by taking the disjunction of each query in the left hand side of those condition-action rule (i.e., $Q_1(\vec{p}) \vee \dots \vee Q_n(\vec{p}) \mapsto \alpha(\vec{p}) \in \Pi$).

Example 3.4. To give an example for KAB, we consider the order processing scenario in Example 2.54. To model such scenario, we specify a KAB $\mathcal{K} = \langle T, A_0, \Gamma, \Pi \rangle$, where T is the same TBox as in Example 2.17, and the initial ABox A_0 is as follows:

$$A_0 = \{\text{ReceivedOrder}(\text{chair}), \text{ApprovedOrder}(\text{table})\}$$

The progression mechanism is then modeled by specifying the set Γ of actions containing the following actions:

1. **approveOrder(x)**, which intuitively approves the order x . Technically it adds a new assertion made by the **ApprovedOrder** concept. Formally it is specified as follows:

$$\text{approveOrder}(x) : \{\text{true} \rightsquigarrow \text{add } \{\text{ApprovedOrder}(x)\}\}$$

2. **prepareOrders()**, which intuitively prepares several things that are needed for further processing steps of each approved order. For each approved order, this action prepares the design of the corresponding order by calling an external service **GETDESIGN(x)**. Moreover, this action also retrieves the information about the corresponding designer by calling a service **GETDESIGNER(x)**, and assign the assembling location for the corresponding order by calling a service **ASSIGNASSEMBLINGLOC(x)**. Formally it is specified as follows:

$$\begin{aligned} \text{prepareOrders}() : \{ \\ & [\text{ApprovedOrder}(x)] \rightsquigarrow \\ & \quad \text{add } \{ \\ & \quad \quad \text{designedBy}(x, \text{GETDESIGNER}(x)), \\ & \quad \quad \text{Designer}(\text{GETDESIGNER}(x)), \\ & \quad \quad \text{hasDesign}(x, \text{GETDESIGN}(x)), \\ & \quad \quad \text{hasAssemblingLoc}(x, \text{ASSIGNASSEMBLINGLOC}(x)) \\ & \quad \} \\ & \} \end{aligned}$$

3. **assembleOrders()**, which represents the step of assembling several components into the corresponding ordered furniture. This action also acquires the information about the assembler and the assembling location by calling external service calls **GETASSEMBLER(x)** and **GETASSEMBLINGLOC(x)** respectively. Formally this action is specified as follows:

$$\begin{aligned} \text{assembleOrders}() : \{ \\ & [\text{ApprovedOrder}(x) \wedge \exists y. \text{hasDesign}(x, y)] \rightsquigarrow \\ & \quad \text{add } \{ \\ & \quad \quad \text{AssembledOrder}(x), \\ & \quad \quad \text{assembledBy}(x, \text{GETASSEMBLER}(x)), \\ & \quad \quad \text{Assembler}(\text{GETASSEMBLER}(x)), \\ & \quad \quad \text{hasAssemblingLoc}(x, \text{GETASSEMBLINGLOC}(x)) \\ & \quad \}, \\ & \quad \text{del } \{ \text{ApprovedOrder}(x) \} \\ & \} \end{aligned}$$

4. **checkAssembledOrders()**, which model the quality check process for each assembled order. This action calls an external service **GETQUALITYCONTROLLER(x)**

in order to obtain the quality controller officer who performs the task. Formally it is specified as follows:

```
checkAssembledOrders() : {
  [AssembledOrder( $x$ )]  $\rightsquigarrow$ 
    add {
      checkedBy( $x$ , GETQUALITYCONTROLLER( $x$ )),
      QualityController(GETQUALITYCONTROLLER( $x$ ))
    }
}
```

5. `deliverOrders()`, which delivers each assembled order that has passed the quality control process. Technically, it changes the status of an assembled order that has been delivered into delivered order. Formally this action is specified as follows:

```
deliverOrders() : {
  [AssembledOrder( $x$ )  $\wedge$   $\exists y$ .checkedBy( $x$ ,  $y$ )]  $\rightsquigarrow$ 
    add {DeliveredOrder( $x$ )}, del {AssembledOrder( $x$ )}
}
```

Furthermore, the set Π of condition action rules is specified as follows:

- $[ReceivedOrder(x)] \mapsto approveOrder(x)$,
- $[\exists x.ApprovedOrder(x)] \mapsto prepareOrders()$,
- $[\exists xy.ApprovedOrder(x) \wedge hasDesign(x, y)] \mapsto assembleOrders()$,
- $[\exists x.AssembledOrder(x)] \mapsto checkAssembledOrders()$,
- $[\exists xy.AssembledOrder(x) \wedge checkedBy(x, y)] \mapsto deliverOrders()$.

The intuition for each condition action rules above is consecutively presented below:

- the first rule states that if there exists a received order with ID x , then we can fire the execution of action `approveOrder/1` with the argument x (i.e., `approveOrder(x)`).
- The second rule says that we can execute the action `prepareOrders/0` in case there exists at least one approved order.
- Next, the third rule encodes the condition where the execution of action `assembleOrders/0` can be fired in case there exists an approved order that already has a design.
- The fourth rule indicates that if there exists an assembled order, we can execute the action `checkAssembledOrders/0`
- Finally, the last rule specifies that whenever there exists an assembled order that has been checked, we can execute the action `deliverOrders/0`.

3.2 KABs Standard Execution Semantics

The *standard execution semantics* of a KAB \mathcal{K} is given in terms of a possibly infinite-state *transition system* whose states are labeled by knowledge bases and where tran-

sitions represent the execution of actions. Such a transition system represents all possible computations that the actions can do on the knowledge base starting from the initial knowledge base (with the corresponding initial ABox).

During the execution, an action can issue service calls. In this thesis, we assume that the semantics of service calls is *deterministic*, i.e., along a run of the system, whenever a service is called with the same input parameters, it will return the same value. To enforce this semantics, similar to DCDS (see Section 2.7), the transition system of a KAB remembers the results of previous service calls in a service call map (see Definition 2.55) that is embedded as a part of the transition system state.

Technically, to provide the semantics of KABs, we consider KB transition systems (as in Definition 2.45), i.e., transition systems of the form $\langle \Delta, T, \Sigma, s_0, abox, \Rightarrow \rangle$, where:

- T is a $DL-Lite_A$ TBox;
- Σ is a (possibly infinite) set of states;
- $s_0 \in \Sigma$ is the initial state;
- $abox$ is a function that, given a state $s \in \Sigma$, returns an ABox associated to s ;
- $\Rightarrow \subseteq \Sigma \times \Sigma$ is a transition relation between pairs of states.

In addition, to realize the deterministic service calls semantics, each state $s \in \Sigma$ of the transition system is defined as a tuple $\langle A, m \rangle$, where A is an ABox and m is a service call map. Later on, a state of the form $\langle A, m \rangle$ is often also called a *KAB state*.

KAB Action
Execution Semantics

In the following, we provide the semantics of an *action execution* by defining when an action can be executed and how to construct the result of an action execution.

Executability of
an Action

Definition 3.5 (Executability of an Action). Let $\mathcal{K} = \langle T, A_0, \Gamma, \Pi \rangle$ be a KAB. Given an ABox A , an action $\alpha \in \Gamma$ of the form $\alpha(\vec{p}) : \{e_1, \dots, e_m\}$, and a *parameter substitution* σ which substitutes the parameters \vec{p} with values taken from Δ . We say that α is *executable in A with a parameter substitution σ* , if there exists a condition-action rule $Q(\vec{p}) \mapsto \alpha(\vec{p}) \in \Pi$ such that $\text{CERT}(Q\sigma, T, A)$ is true. ■

Similar to DCDS (as in Section 2.7), with a little abuse of the definition, we sometimes say that an action α is *executable in a state s with a parameter substitution σ* if $s = \langle A, m \rangle$ and α is executable in A with a parameter substitution σ . Additionally, given an action $\alpha(p_1, \dots, p_n) : \{e_1, \dots, e_m\}$, and an ABox A , We say that σ is a *legal parameter assignment* for α in A , if α is executable in A with a parameter substitution σ . Furthermore, we write $\alpha\sigma$ to denote a *grounded action* that is obtained by applying a legal parameter assignment σ to each $e \in \text{EFF}(\alpha)$ (i.e., substituting each occurrence of action parameter p_i (for $i \in \{1, \dots, n\}$) in e with a constant in Δ based on the substitution σ).

Legal Parameter
Assignment and
Grounded Action

Example 3.6. Continuing our running example in Example 3.4, the action `approveOrder/1` is executable in the state $s_0 = \langle A_0, \emptyset \rangle$ with a parameter substitution σ , where σ substitutes the action parameter of `approveOrder/1` into `chair`. This is the case because we have that the query in the left hand side of the condition-action rule

$$[\text{ReceivedOrder}(x)] \mapsto \text{approveOrder}(x),$$

is successfully evaluated and give an answer `chair` (i.e., $\sigma(x) = \text{chair}$).

Now we proceed to define a set of atoms to be added/deleted by a grounded action $\alpha\sigma$.

Definition 3.7 (Set of Atoms to be Added). Let $\mathcal{K} = \langle T, A_0, \Gamma, \Pi \rangle$ be a KAB. Given an ABox A , an action $\alpha \in \Gamma$ of the form $\alpha(\vec{p}) : \{e_1, \dots, e_m\}$ with $e_i = [q_i^+] \wedge Q_i^- \rightsquigarrow \mathbf{add} F_i^+, \mathbf{del} F_i^-$, and a legal parameter assignment σ for α in A . We define a *set of atoms to be added by $\alpha\sigma$ w.r.t. A* as follows:

Set of Atoms to be Added by an Action

$$\text{ADD}(T, A, \alpha\sigma) = \bigcup_{([q^+] \wedge Q^- \rightsquigarrow \mathbf{add} F^+, \mathbf{del} F^-) \text{ in Eff}(\alpha)} \bigcup_{\rho \in \text{CERT}([q^+] \wedge Q^- \sigma, T, A)} F^+ \sigma \rho$$

■

Definition 3.8 (Set of Atoms to be Deleted). Let $\mathcal{K} = \langle T, A_0, \Gamma, \Pi \rangle$ be a KAB. Given an ABox A , an $\alpha \in \Gamma$ of the form $\alpha(\vec{p}) : \{e_1, \dots, e_m\}$ with $e_i = [q_i^+] \wedge Q_i^- \rightsquigarrow \mathbf{add} F_i^+, \mathbf{del} F_i^-$, and a legal parameter assignment σ for α in A . We define a *set of atoms to be deleted by $\alpha\sigma$ w.r.t. A* as follows:

Set of Atoms to be Deleted by an Action

$$\text{DEL}(T, A, \alpha\sigma) = \bigcup_{([q^+] \wedge Q^- \rightsquigarrow \mathbf{add} F^+, \mathbf{del} F^-) \text{ in Eff}(\alpha)} \bigcup_{\rho \in \text{CERT}([q^+] \wedge Q^- \sigma, T, A)} F^- \sigma \rho$$

■

The execution result of a grounded action $\alpha\sigma$ is then captured by a function $\text{DO}(T, A, \alpha\sigma)$ which is formally defined as follows:

Definition 3.9 (Computation of KAB Action Execution Result). Let $\mathcal{K} = \langle T, A_0, \Gamma, \Pi \rangle$ be a KAB. Given an ABox A , an action $\alpha \in \Gamma$ of the form $\alpha(\vec{p}) : \{e_1, \dots, e_m\}$ with $e_i = [q_i^+] \wedge Q_i^- \rightsquigarrow \mathbf{add} F_i^+, \mathbf{del} F_i^-$, and a legal parameter assignment σ for α in A . The *execution result of $\alpha\sigma$ in A* is computed by function $\text{DO}(T, A, \alpha\sigma)$ as follows:

Computation of KAB Action Execution Result

$$\text{DO}(T, A, \alpha\sigma) = (A \setminus \text{DEL}(T, A, \alpha\sigma)) \cup (\text{ADD}(T, A, \alpha\sigma))$$

where $e = [q^+] \wedge Q^- \rightsquigarrow \mathbf{add} F^+, \mathbf{del} F^-$.

■

Radically different from the KABs in [121] (also different from DCDSs), during an action execution, instead of dropping the whole ABox and constructing a new one as the result of the action execution, in this KABs the actions only update the corresponding current ABox (i.e., add/delete assertions from the current ABox). Notice that in this sense such actions are similar to STRIPS style actions [102] where the additions are assumed to have higher priority than deletions, and those “facts” that are not affected by the action execution stay the same. Hence, unlike in DCDSs, we do not need to specify both the things that are changes and the things that are not changes. More precisely, the definition above intuitively says that the execution result of α is obtained by first deleting from A the assertions that is obtained from the grounding of the atoms in F^- and then adds the new assertions that is obtained from the grounding of the atoms in F^+ . The grounding of the atoms in F^+ and F^- are obtained from all the certain answers of the query $[q^+] \wedge Q^-$ over $\langle T, A \rangle$.

The result of $\text{DO}(T, A, \alpha\sigma)$ is in general not a proper ABox, because it could contain (ground) skolem terms, attesting that in order to produce the ABox, some service calls

Service Call Evaluation have to be issued. We denote by $\text{CALLS}(\text{DO}(T, A, \alpha\sigma))$ the set of such ground service calls, and by $\text{EVAL}(T, A, \alpha\sigma)$ the set of substitutions that replace such calls with concrete values taken from Δ . Specifically, $\text{EVAL}(T, A, \alpha\sigma)$ is defined as

$$\text{EVAL}(T, A, \alpha\sigma) = \{\theta \mid \theta \text{ is a total function } \theta : \text{CALLS}(\text{DO}(T, A, \alpha\sigma)) \rightarrow \Delta\}.$$

Given a KAB $\mathcal{K} = \langle T, A_0, \Gamma, \Pi \rangle$, we employ $\text{DO}()$ and $\text{EVAL}()$ to define a transition relation $\text{EXEC}_{\mathcal{K}}$ connecting two states through an action execution as follows.

Definition 3.10 (KAB Transition Relation $\text{EXEC}_{\mathcal{K}}$).

KAB Transition Relation Let $\mathcal{K} = \langle T, A_0, \Gamma, \Pi \rangle$ be a KAB. Given a state $s = \langle A, m \rangle$, a state $s' = \langle A', m' \rangle$, an action $\alpha \in \Gamma$, and a substitution σ . We have $\langle \langle A, m \rangle, \alpha\sigma, \langle A', m' \rangle \rangle \in \text{EXEC}_{\mathcal{K}}$ if the following holds:

1. α is executable in s with a legal parameter assignment σ ;
2. there exists $\theta \in \text{EVAL}(T, A, \alpha\sigma)$ such that θ and m “agree” on the common skolem terms in their domains, in order to realize the deterministic service call semantics;
3. $A' = \text{DO}(T, A, \alpha\sigma)\theta$;
4. $m' = m \cup \theta$ (i.e., updating the history of issued service calls).

■

For easiness of reading, when $\langle \langle A, m \rangle, \alpha\sigma, \langle A', m' \rangle \rangle \in \text{EXEC}_{\mathcal{K}}$, we equivalently write

$$\langle A, m \rangle \xrightarrow{\alpha\sigma, \mathcal{K}} \langle A', m' \rangle.$$

When it is clear from the context, we also often omit \mathcal{K} and just write $\langle A, m \rangle \xrightarrow{\alpha\sigma} \langle A', m' \rangle$.

The transition system $\Upsilon_{\mathcal{K}}$ of KAB \mathcal{K} , which provide the standard execution semantics of \mathcal{K} , is then formally defined as follows:

KABs Standard Transition System **Definition 3.11** (KABs Standard Transition System). Given a KAB $\mathcal{K} = \langle T, A_0, \Gamma, \Pi \rangle$, the standard transition system $\Upsilon_{\mathcal{K}}$ is defined as $\langle \Delta, T, \Sigma, s_0, \text{abox}, \Rightarrow \rangle$ where

- $s_0 = \langle A_0, \emptyset \rangle$, and
- Σ and \Rightarrow are defined by simultaneous induction as the smallest sets satisfying the following properties:

1. $s_0 \in \Sigma$;
2. if $\langle A, m \rangle \in \Sigma$, then for all actions $\alpha \in \Gamma$, for all substitutions σ for the parameters of α and for all $\langle A', m' \rangle$ such that

$$\text{a) } \langle A, m \rangle \xrightarrow{\alpha\sigma, \mathcal{K}} \langle A', m' \rangle \text{ and}$$

$$\text{b) } A' \text{ is } T\text{-consistent,}$$

$$\text{we have } \langle A', m' \rangle \in \Sigma, \text{ and } \langle A, m \rangle \Rightarrow \langle A', m' \rangle.$$

■

Intuitively, the standard execution semantics for a KAB $\mathcal{K} = \langle T, A_0, \Gamma, \Pi \rangle$ is obtained starting from A_0 by nondeterministically applying every executable actions with corresponding legal parameter assignments, and considering each possible value returned by applying the involved service calls. The executability of an action with fixed parameters does not only depend on the set of condition-action rules Π , but also on the T -consistency of the ABox produced by the execution of the action: if the resulting ABox is T -inconsistent, the action is considered as non executable with the chosen parameters.

Example 3.12. Continuing our running example in Example 3.4, the construction of transition system $\mathcal{T}_{\mathcal{K}}$ of KAB \mathcal{K} in Example 3.4 is started from the initial state $s_0 = \langle A_0, \emptyset \rangle$ where

$$A_0 = \{\text{ReceivedOrder}(\text{chair}), \text{ApprovedOrder}(\text{table})\}.$$

An example of a successor of state s_0 is a state $s_1 = \langle A_1, m_1 \rangle$, where

$$\begin{aligned} A_1 &= \{\text{ApprovedOrder}(\text{chair}), \text{ApprovedOrder}(\text{table})\}, \\ m_1 &= \emptyset \end{aligned}$$

and s_1 is obtained from the execution of action `approveOrder/1` with the argument `chair`. The action `approveOrder/1` is executable with argument `chair` in the state s_0 since we have that the query in the left hand side of the condition-action rule

$$[\text{ReceivedOrder}(x)] \mapsto \text{approveOrder}(x),$$

is successfully evaluated and give an answer `chair`. As the result of executing this action, we now have an approved order of chair in state s_1 .

3.3 Verification of KABs

The interesting reasoning task in KABs is to verify whether the transition system of a given KAB satisfies temporal properties of interest, specified in some first-order temporal logic. To specify the temporal properties to be verified over KABs, the work in [121] uses the temporal logic $\mu\mathcal{L}_A^{\text{EQL}}$ (see Section 2.6.1). The verification problem of $\mu\mathcal{L}_A^{\text{EQL}}$ properties over KABs is then formally stated as follows:

Definition 3.13 (Verification of a $\mu\mathcal{L}_A^{\text{EQL}}$ Property over a KAB). Given a KAB $\mathcal{K} = \langle T, A_0, \Gamma, \Pi \rangle$ and a closed $\mu\mathcal{L}_A^{\text{EQL}}$ formula Φ over $\langle T, A_0 \rangle$. Let $\mathcal{T}_{\mathcal{K}}$ be the transition system of \mathcal{K} , the verification of a $\mu\mathcal{L}_A^{\text{EQL}}$ formula Φ over \mathcal{K} is a problem to check whether $\mathcal{T}_{\mathcal{K}} \models \Phi$. ■

Verification of a $\mu\mathcal{L}_A^{\text{EQL}}$ Formula over a KAB

We also say a KAB \mathcal{K} satisfies a closed $\mu\mathcal{L}_A^{\text{EQL}}$ formula Φ , if $\mathcal{T}_{\mathcal{K}} \models \Phi$. The definition above intuitively said that given a KAB \mathcal{K} and a closed $\mu\mathcal{L}_A^{\text{EQL}}$ formula Φ , the problem of verifying Φ over \mathcal{K} is a problem to check whether Φ holds in the initial state of $\mathcal{T}_{\mathcal{K}}$. From this moment, we assume that the $\mu\mathcal{L}_A^{\text{EQL}}$ properties to be verified over KABs $\mathcal{K} = \langle T, A_0, \Gamma, \Pi \rangle$ are closed $\mu\mathcal{L}_A^{\text{EQL}}$ formulas over $\langle T, A_0 \rangle$.

Here we solve the problem of $\mu\mathcal{L}_A^{\text{EQL}}$ properties verification over KABs by reducing it into the problem of $\mu\mathcal{L}_A$ properties verification over DCDSs. The idea of the reduction is similar to the work in [146], which shows a reduction from Data-Aware Commitment-Based Multiagent Systems into DCDSs. To reduce such problem, here we do the following:

1. We define a generic translation τ_{dcds} , that given a KAB \mathcal{K} , produces a DCDS $\tau_{dcds}(\mathcal{K})$, and
2. We extend the perfect reformulation algorithm into $\mu\mathcal{L}_A^{\text{EQL}}$ formula by simply applying the algorithm only to each query in the corresponding $\mu\mathcal{L}_A^{\text{EQL}}$ formula and keeping the other parts of the formula unaltered.
3. We show that a KAB \mathcal{K} satisfies a certain $\mu\mathcal{L}_A^{\text{EQL}}$ property Φ if and only if its corresponding DCDS $\tau_{dcds}(\mathcal{K})$ (obtained from \mathcal{K} via τ_{dcds}) satisfies a $\mu\mathcal{L}_A$ property Φ' that is obtained using the extended perfect reformulation algorithm for $\mu\mathcal{L}_A^{\text{EQL}}$ (i.e., by rewriting each query in Φ w.r.t. the TBox in the given KAB \mathcal{K} using the perfect reformulation algorithm).

3.3.1 Translating KABs into DCDSs

In this section we define a generic translation which transform any KABs into DCDSs with the aim of reducing the $\mu\mathcal{L}_A^{\text{EQL}}$ verification problem over KABs into the $\mu\mathcal{L}_A$ verification problem over DCDSs. Towards this goal, we first introduce several preliminaries as follows.

Equality Between a Database Instance and an ABox

Definition 3.14 (Equality Between a Database Instance and an ABox). Given a database instance \mathbf{I} which conforms to schema \mathcal{R} , and an ABox A over $\text{voc}(T)$, we say that A is equal to \mathbf{I} , denoted by $A = \mathbf{I}$, if the following hold

- a concept name $N \in \text{voc}(T)$ if and only if a relation schema $N \in \mathcal{R}$,
- a role name $P \in \text{voc}(T)$ if and only if a relation schema $P \in \mathcal{R}$,
- a concept assertion $N(c) \in A$ if and only if a database fact $N(c) \in \mathbf{I}$,
- a role assertion $P(c_1, c_2) \in A$ if and only if a database fact $P(c_1, c_2) \in \mathbf{I}$.

■

Set of Deletion Effects

Definition 3.15 (Set of Deletion Effects For Concept Assertion). Given a KAB $\mathcal{K} = \langle T, A_0, \Gamma, \Pi \rangle$, and an action of the form $\alpha(\vec{p}) : \{e_1, \dots, e_m\}$, we define a *set of deletion effects for concept assertions* made by concept name $N \in \text{voc}(T)$ w.r.t. α as a set of effects constructed as follows: we have

$$[q^+] \wedge Q^- \rightsquigarrow \mathbf{del} \{N(t)\} \in \text{EFF}_d(N, \alpha)$$

if there exists an effect

$$[q^+] \wedge Q^- \rightsquigarrow \mathbf{add} F^+, \mathbf{del} F^- \in \text{EFF}(\alpha)$$

such that $N(t) \in F^-$, where t can be a free variable of $[q^+]$, an action parameter (i.e., among \vec{p}), or a constant in $\text{ADOM}(A_0)$. ■

The case for role assertions (i.e., $\text{EFF}_d(P, \alpha)$) is defined similarly as above.

Having all necessary preliminaries in hand, we are ready to define a translation τ_{dcds} that, given a KAB \mathcal{K} , produces DCDS $\tau_{dcds}(\mathcal{K})$ as follows.

Definition 3.16 (Translation From KABs to DCDSs). A *translation* τ_{dcds} is a translation that takes a KAB $\mathcal{K} = \langle T, A_0, \Gamma, \Pi \rangle$ as input and produces DCDS $\tau_{dcds}(\mathcal{K}) = \langle D, P \rangle$, where:

- $D = \langle \mathcal{R}, \mathbf{I}_0, \mathcal{E} \rangle$ is a data component, where:
 - \mathcal{R} is obtained as follows:
 - * for each concept name $N \in \text{voc}(T)$, we have $N \in \mathcal{R}$ with arity 1,
 - * for each role name $P \in \text{voc}(T)$, we have $P \in \mathcal{R}$ with arity 2,
 - $\mathbf{I}_0 = A_0$,
 - $\mathcal{E} = \{Q_{\text{unsatFOL}}^T \rightarrow \text{false}\}$, where Q_{unsatFOL}^T is an FOL query defined in Definition 2.43. Intuitively, here we encode the constraints in the TBox T into the equality constraints \mathcal{E} in DCDS.
- $P = \langle \mathcal{A}, \varrho \rangle$ is a process component, where
 - \mathcal{A} is obtained as follows: for each $\alpha(\vec{p}) \in \Gamma$, we have $\alpha'(\vec{p}) \in \mathcal{A}$ that is constructed as follows:
 - (1) For each effect $[q^+] \wedge Q^- \rightsquigarrow \mathbf{add} F^+, \mathbf{del} F^- \in \text{EFF}(\alpha)$, we have $\text{rew}([q^+] \wedge Q^-, T) \rightsquigarrow F^+ \in \text{EFF}(\alpha')$,
 - (2) For each concept name $N \in \text{voc}(T)$, we have an effect of the form

$$N(w) \wedge \bigwedge_{e \in \text{EFF}_d(N, \alpha)} (\text{rew}([q^+] \wedge Q^-, T) \wedge \neg(w = t)) \rightsquigarrow \{N(w)\}$$

in $\text{EFF}(\alpha')$, where

- $e = [q^+] \wedge Q^- \rightsquigarrow \mathbf{del} \{N(t)\} \in \text{EFF}_d(N, \alpha)$,
- w is a variable, and additionally it is neither an action parameter nor a free variable of any queries $[q^+]$ in any $e \in \text{EFF}_d(N, \alpha)$.

Intuitively, the effects constructed above, preserve all concept assertions that are not deleted by any deletion effect.

- (3) We repeat similar construction, as in the (2), for each role name $P \in \text{voc}(T)$.
- ϱ is obtained as follows: for each condition-action rule $Q \mapsto \alpha \in \Pi$, we have $Q' \mapsto \alpha' \in \varrho$, where $Q' = \text{rew}(Q, T)$, and $\alpha' \in \mathcal{A}$ is obtained from $\alpha \in \Gamma$ as above.

■

In the following, we show several interesting properties of the translation τ_{dcds} that will be used later to reduce the problem of $\mu\mathcal{L}_A^{\text{EQL}}$ verification over KABs into the problem of $\mu\mathcal{L}_A$ verification over DCDSs. First, we show that the computation result of KAB action execution α over a certain state s_k and the computation result of DCDS action execution α' (which is obtained from α via τ_{dcds}) over a certain state s_d produce a same result provided that the corresponding ABox in s_k and database instance in s_d are equal.

Lemma 3.17. *Let $\mathcal{K} = \langle T, A_0, \Gamma, \Pi \rangle$ be a KAB with transition system $\Upsilon_{\mathcal{K}}$, $\tau_{dcds}(\mathcal{K}) = \langle D, P \rangle$ be a DCDS obtained from \mathcal{K} through translation τ_{dcds} where $P = \langle \mathcal{A}, \varrho \rangle$. Additionally, let $\Upsilon_{\tau_{dcds}(\mathcal{K})}$ be transition system of $\tau_{dcds}(\mathcal{K})$. Consider a state $s = \langle A, m \rangle$ of $\Upsilon_{\mathcal{K}}$, a state $s' = \langle \mathbf{I}, m_d \rangle$ of $\Upsilon_{\tau_{dcds}(\mathcal{K})}$, an action $\alpha \in \Gamma$, an action $\alpha' \in \mathcal{A}$ that is obtained from α as in the definition of translation τ_{dcds} , and a substitution σ that is a legal parameter assignment for α in s and also a legal parameter assignment for α' in s' . If $A = \mathbf{I}$, then $\text{DO}(T, A, \alpha\sigma) = \text{DO}_{\text{DCDS}}(\mathbf{I}, \alpha'\sigma)$.*

Proof. We have to show that

- (1) $N(t_1) \in \text{DO}(T, A, \alpha\sigma)$ if and only if $N(t_1) \in \text{DO}_{\text{DCDS}}(\mathbf{I}, \alpha'\sigma)$
- (2) $P(t_1, t_2) \in \text{DO}(T, A, \alpha\sigma)$ if and only if $P(t_1, t_2) \in \text{DO}_{\text{DCDS}}(\mathbf{I}, \alpha'\sigma)$

where N is a concept name, P is a role name, and t_1 (resp. t_2) is either a constant in Δ or a skolem term.

Proof for Case (1): by Definition 3.9, if $N(t) \in \text{DO}(T, A, \alpha\sigma)$ then we have either

- (a) $N(t_1) \in \text{ADD}(T, A, \alpha\sigma)$ (No matter whether $N(t_1) \in A$ or $N(t_1) \notin A$, and also no matter whether $N(t_1) \in \text{DEL}(T, A, \alpha\sigma)$ or $N(t_1) \notin \text{DEL}(T, A, \alpha\sigma)$).
- (b) $N(t_1) \in A$ and $N(t_1) \notin \text{DEL}(T, A, \alpha\sigma)$, or

Case (a): By Definition 3.7, if $N(t_1) \in \text{ADD}(T, A, \alpha\sigma)$ then there exists an effect

$$[q^+] \wedge Q^- \rightsquigarrow \mathbf{add} F^+, \mathbf{del} F^- \in \text{EFF}(\alpha)$$

s.t. $N(x) \in F^+$ where x is either an action parameter, a constant in $\text{ADOM}(A_0)$, a free variable in $[q^+]$ or a skolem terms formed by applying a function $f \in \mathcal{F}$ to either constants in $\text{ADOM}(A_0)$, action parameters, or free variables of $[q^+]$.

Moreover, the following hold:

- if x is a constant in $\text{ADOM}(A_0)$, then $x = t_1$
- if x is an action parameter, then $x/t_1 \in \sigma$
- if x is a free variable in $[q^+]$, then $x/t_1 \in \rho$, where $\rho \in \text{CERT}([q^+] \wedge Q^- \sigma, T, A)$
- if x is a skolem term of the form $f(\vec{v})$, then we have either $(f(\vec{v})\sigma)\rho = t_1$, where $\rho \in \text{CERT}([q^+] \wedge Q^- \sigma, T, A)$

Now, recall that by the definition of τ_{dcds} (Definition 3.16), since

$$[q^+] \wedge Q^- \rightsquigarrow \mathbf{add} F^+, \mathbf{del} F^- \in \text{EFF}(\alpha)$$

then we have

$$\text{rew}([q^+] \wedge Q^-, T) \rightsquigarrow F^+ \in \text{EFF}(\alpha').$$

Additionally, since $A = \mathbf{I}$, by Theorem 2.39, we have

$$\text{cert}([q^+] \wedge Q^- \sigma, T, A) = \text{ANS}(\text{rew}([q^+] \wedge Q^-, T)\sigma, \mathbf{I}).$$

Thus, by Definition 2.58 it is easy to see that $N(t_1) \in \text{DO}_{\text{DCDS}}(\mathbf{I}, \alpha'\sigma)$.

Case (b): If $N(t_1) \in A$ and $N(t_1) \notin \text{DEL}(T, A, \alpha\sigma)$, then we have

1. there exists a substitution ρ such that $\rho \in \text{CERT}([N(w)], T, A)$, and $[w/t_1] \in \rho$.
2. there does not exists $[q^+] \wedge Q^- \rightsquigarrow \mathbf{del} \{N(t)\} \in \text{EFF}_d(N, \alpha)$ such that there exists $\rho \in \text{CERT}([q^+] \wedge Q^-, T, A)$ and $t/t_1 \in \rho$.

Now, recall that by the definition of τ_{dcds} (Definition 3.16), For each concept name $N \in \text{VOC}(T)$, we have an effect of the form

$$N(w) \wedge \bigwedge_{e \in \text{EFF}_d(N, \alpha)} (\text{rew}([q^+] \wedge Q^-, T) \wedge \neg(w = t)) \rightsquigarrow \{N(w)\}$$

in $\text{EFF}(\alpha')$, where

- $e = [q^+] \wedge Q^- \rightsquigarrow \mathbf{del} \{N(t)\} \in \text{EFF}_d(N, \alpha)$,
- w is a variable, and additionally it is neither an action parameter nor a free variable of any queries $[q^+]$ in any $e \in \text{EFF}_d(N, \alpha)$.

Additionally, since $A = \mathbf{I}$, by Theorem 2.39, we have

1. there exists a substitution ρ such that $\rho \in \text{ANS}([N(w)], \mathbf{I})$, and $w/t_1 \in \rho$.
2. there does not exist a query $\text{rew}([q^+] \wedge Q^-, T)$ (where $[q^+] \wedge Q^- \rightsquigarrow \text{del } \{N(t)\} \in \text{EFF}_d(N, \alpha)$) such that $\rho \in \text{ANS}(\text{rew}([q^+] \wedge Q^-, T), \mathbf{I})$ and $t/t_1 \in \rho$.

Thus, by Definition 2.58 we have $N(t_1) \in \text{DO}_{\text{DCDS}}(\mathbf{I}, \alpha'\sigma)$.

The proof for the other direction for case (1) can be shown similarly. Moreover, the proof for case (2) can be done similarly as the case (1). \square

As a consequence of Lemma 3.17, in the following we show that the set of substitutions that replace service calls $\text{EVAL}(T, A, \alpha\sigma)$ is equal to $\text{EVAL}(\mathbf{I}, \alpha'\sigma)$ provided that (i) $A = \mathbf{I}$, (ii) α' is obtained from α through τ_{dcds} and both of them are grounded with the same legal parameter assignment σ .

Lemma 3.18. *Let $\mathcal{K} = \langle T, A_0, \Gamma, \Pi \rangle$ be a KAB with transition system $\Upsilon_{\mathcal{K}}$, $\tau_{\text{dcds}}(\mathcal{K}) = \langle D, P \rangle$ be a DCDS obtained from \mathcal{K} through translation τ_{dcds} where $P = \langle \mathcal{A}, \varrho \rangle$. Additionally, let $\Upsilon_{\tau_{\text{dcds}}(\mathcal{K})}$ be transition system of $\tau_{\text{dcds}}(\mathcal{K})$. Consider a state $s = \langle A, m \rangle$ of $\Upsilon_{\mathcal{K}}$, a state $s' = \langle \mathbf{I}, m_d \rangle$ of $\Upsilon_{\tau_{\text{dcds}}(\mathcal{K})}$, an action $\alpha \in \Gamma$, an action $\alpha \in \mathcal{A}$ obtained from α as in the definition of translation τ_{dcds} , and a substitution σ that is a legal parameter assignment for α in s as well as a legal parameter assignment for α' in s' . If $A = \mathbf{I}$, then $\text{EVAL}(T, A, \alpha\sigma) = \text{EVAL}(\mathbf{I}, \alpha'\sigma)$.*

Proof. By Lemma 3.17, we have $\text{DO}(T, A, \alpha\sigma) = \text{DO}_{\text{DCDS}}(\mathbf{I}, \alpha'\sigma)$. Therefore by the definition of $\text{EVAL}(T, A, \alpha\sigma)$ and $\text{EVAL}(\mathbf{I}, \alpha'\sigma)$, it is easy to see that $\text{EVAL}(T, A, \alpha\sigma) = \text{EVAL}(\mathbf{I}, \alpha'\sigma)$. \square

Last, we show that an equality constraints \mathcal{E} , that is obtained through τ_{dcds} , encode the same constraints as in the TBox T (of the corresponding KAB) that is needed for satisfiability check. As a consequence, given an ABox A and a database instance \mathbf{I} such that $A = \mathbf{I}$, if A is T -consistent then \mathbf{I} satisfies \mathcal{E} , and vice versa.

Lemma 3.19. *Let $\mathcal{K} = \langle T, A_0, \Gamma, \Pi \rangle$ be a KAB with transition system $\Upsilon_{\mathcal{K}}$, $\tau_{\text{dcds}}(\mathcal{K}) = \langle D, P \rangle$ be a DCDS obtained from \mathcal{K} through translation τ_{dcds} , where $D = \langle \mathcal{R}, \mathbf{I}_0, \mathcal{E} \rangle$. Additionally, let $\Upsilon_{\tau_{\text{dcds}}(\mathcal{K})}$ be transition system of $\tau_{\text{dcds}}(\mathcal{K})$. Consider a state $s = \langle A, m \rangle$ of $\Upsilon_{\mathcal{K}}$, a state $s' = \langle \mathbf{I}, m_d \rangle$ of $\Upsilon_{\tau_{\text{dcds}}(\mathcal{K})}$, such that $A = \mathbf{I}$, then we have \mathbf{I} satisfies \mathcal{E} if and only if A is T -consistent*

Proof. Since A is T -consistent, by Theorem 2.44, we have $\text{ANS}(Q_{\text{unsatFOL}}^T, A) = \text{false}$. Then, since, $\mathbf{I} = A$, we have $\text{ANS}(Q_{\text{unsatFOL}}^T, \mathbf{I}) = \text{false}$. Therefore, by Definition 2.48, and since $\mathcal{E} = \{Q_{\text{unsatFOL}}^T \rightarrow \text{false}\}$, we have \mathbf{I} satisfies \mathcal{E} . \square

3.3.2 Rewriting $\mu\mathcal{L}_A^{\text{EQL}}$ Formulas

We extend the perfect reformulation algorithm to $\mu\mathcal{L}_A^{\text{EQL}}$ formulas as follows.

Perfect Reformulation
of $\mu\mathcal{L}_A^{\text{EQL}}$ Formula

Definition 3.20 (Perfect Reformulation of $\mu\mathcal{L}_A^{\text{EQL}}$ Formula). Given a KB $\langle T, A \rangle$, and a $\mu\mathcal{L}_A^{\text{EQL}}$ formula Φ over $\langle T, A \rangle$, a *perfect reformulation* of Φ w.r.t. T is $\Phi' = \text{rew}(\Phi, T)$, where $\text{rew}(\Phi, T)$ is inductively defined as follows:

$$\text{rew}(\Phi, T) = \begin{cases} \text{rew}(Q, T) & \text{if } \Phi = Q \\ \text{rew}(\Psi_1, T) \vee \text{rew}(\Psi_2, T) & \text{if } \Phi = \Psi_1 \vee \Psi_2 \\ \exists x. \text{rew}(\Psi, T) & \text{if } \Phi = \exists x. \Psi \\ \langle \neg \rangle \text{rew}(\Psi, T) & \text{if } \Phi = \langle \neg \rangle \Psi \\ \mu Z. \text{rew}(\Psi, T) & \text{if } \Phi = \mu Z. \Psi \end{cases}$$

and $\text{rew}(Q, T)$ is the application of the perfect reformulation algorithm over Q w.r.t. T . ■

The definition above intuitively said that the rewriting of $\mu\mathcal{L}_A^{\text{EQL}}$ formula Φ is obtained by rewriting each query in Φ w.r.t. T using the perfect reformulation algorithm while maintaining the temporal operators unaltered.

3.3.3 Recasting the Verification of KABs into DCDSs

The idea to recast the problem of $\mu\mathcal{L}_A^{\text{EQL}}$ verification over KABs into the problem of $\mu\mathcal{L}_A$ verification over DCDSs is as follows:

1. We define a bisimulation relation namely *KAB-DCDS Bisimulation* (*KD-Bisimulation*) and show some interesting properties of KD-Bisimulation relation which are related to satisfiability of $\mu\mathcal{L}_A^{\text{EQL}}$ formulas.
2. We show that the transition system of a KAB \mathcal{K} and the transition system of its corresponding DCDS $\tau_{dcds}(\mathcal{K})$ (obtained through translation τ_{dcds}) are bisimilar w.r.t. the KD-Bisimulation relation,
3. Making use the ingredients obtained from the point 1 and 2, we show that a KAB \mathcal{K} satisfies a $\mu\mathcal{L}_A^{\text{EQL}}$ property Φ if and only if its corresponding DCDS $\tau_{dcds}(\mathcal{K})$ (obtained through translation τ_{dcds}) satisfies the $\mu\mathcal{L}_A$ property $\text{rew}(\Phi, T)$ (where T is the TBox in the given KAB \mathcal{K}).

We define a KD-Bisimulation relation between a KB transition system and a database transition system as follows.

KAB-DCDS
Bisimulation
(KD-Bisimulation)

Definition 3.21 (KAB-DCDS Bisimulation (KD-Bisimulation)). Let $\mathcal{T}_1 = \langle \Delta, T, \Sigma_1, s_{01}, \text{abox}, \Rightarrow_1 \rangle$ be a KB transition system (see Definition 2.45) and $\mathcal{T}_2 = \langle \Delta, \mathcal{R}, \Sigma_2, s_{02}, \text{db}, \Rightarrow_2 \rangle$ be a database transition system (see Definition 2.46), with $\text{ADOM}(\text{abox}(s_{01})) \subseteq \Delta$, and $\text{ADOM}(\text{db}(s_{02})) \subseteq \Delta$. A *KD-Bisimulation* between \mathcal{T}_1 and \mathcal{T}_2 is a relation $\mathcal{B} \subseteq \Sigma_1 \times \Sigma_2$ such that $\langle s_1, s_2 \rangle \in \mathcal{B}$ implies that:

1. $\text{abox}(s_1) = \text{db}(s_2)$
2. for each s'_1 , if $s_1 \Rightarrow_1 s'_1$ then there exists s'_2 with $s_2 \Rightarrow_2 s'_2$ such that $\langle s'_1, s'_2 \rangle \in \mathcal{B}$.
3. for each s'_2 , if $s_2 \Rightarrow_2 s'_2$ then there exists s'_1 with $s_1 \Rightarrow_1 s'_1$ such that $\langle s'_1, s'_2 \rangle \in \mathcal{B}$. ■

Given a KB transition system $\mathcal{T}_1 = \langle \Delta, T, \Sigma_1, s_{01}, \text{abox}, \Rightarrow_1 \rangle$ and a database transition system $\mathcal{T}_2 = \langle \Delta, \mathcal{R}, \Sigma_2, s_{02}, \text{db}, \Rightarrow_2 \rangle$, we say a state $s_1 \in \Sigma_1$ is *KD-bisimilar* to $s_2 \in \Sigma_2$, written $s_1 \sim_{\text{KD}} s_2$, if there exists a KD-Bisimulation \mathcal{B} between \mathcal{T}_1 and

\mathcal{T}_2 such that $\langle s_1, s_2 \rangle \in \mathcal{B}$. Moreover, a transition system \mathcal{T}_1 is *KD-bisimilar* to \mathcal{T}_2 , written $\mathcal{T}_1 \sim_{\text{KD}} \mathcal{T}_2$, if there exists a KD-Bisimulation \mathcal{B} between \mathcal{T}_1 and \mathcal{T}_2 such that $\langle s_{01}, s_{02} \rangle \in \mathcal{B}$.

Now, we show some important properties of KD-Bisimulation relation w.r.t. satisfiability of $\mu\mathcal{L}_A^{\text{EQL}}$ formulas as follows.

Lemma 3.22. *Consider a KB transition system $\mathcal{T}_1 = \langle \Delta, T, \Sigma_1, s_{01}, \text{abox}, \Rightarrow_1 \rangle$ and a database transition system $\mathcal{T}_2 = \langle \Delta, \mathcal{R}, \Sigma_2, s_{02}, \text{db}, \Rightarrow_2 \rangle$ with $\text{ADOM}(\text{abox}(s_{01})) \subseteq \Delta$, and $\text{ADOM}(\text{db}(s_{02})) \subseteq \Delta$. Consider also two states $s_1 \in \Sigma_1$ and $s_2 \in \Sigma_2$ such that $s_1 \sim_{\text{KD}} s_2$. Then for every (open) $\mu\mathcal{L}_A^{\text{EQL}}$ formula Φ , and for every valuations v_1 and v_2 that assign to each of its free variables a constant $c_1 \in \text{ADOM}(\text{abox}(s_1))$ and $c_2 \in \text{ADOM}(\text{db}(s_2))$, such that $c_1 = c_2$, we have that*

$$\mathcal{T}_1, s_1 \models \Phi v_1 \text{ if and only if } \mathcal{T}_2, s_2 \models \text{rew}(\Phi, T)v_2.$$

Proof. The proof is then organized in three parts:

- (1) We prove the claim for formulae of $\mathcal{L}_A^{\text{EQL}}$, obtained from $\mu\mathcal{L}_A^{\text{EQL}}$ by dropping the predicate variables and the fixpoint constructs. $\mathcal{L}_A^{\text{EQL}}$ corresponds to a first-order variant of the Hennessy Milner logic, and its semantics does not depend on the second-order valuation.
- (2) We extend the results to the infinitary logic obtained by extending $\mathcal{L}_A^{\text{EQL}}$ with arbitrary countable disjunction.
- (3) We recall that fixpoints can be translated into this infinitary logic (cf. [170]), thus proving that the theorem holds for $\mu\mathcal{L}_A^{\text{EQL}}$.

Proof for $\mathcal{L}_A^{\text{EQL}}$. We proceed by induction on the structure of Φ , without considering the case of predicate variable and of fixpoint constructs, which are not part of $\mathcal{L}_A^{\text{EQL}}$.

Base case:

- ($\Phi = Q$). Since $s_1 \sim_{\text{KD}} s_2$, we have $\text{abox}(s_1) = \text{db}(s_2)$. Hence, by Theorem 2.40, for every valuations v_1 and v_2 that assign to each of its free variables a constant $c_1 \in \text{ADOM}(\text{abox}(s_1))$ and $c_2 \in \text{ADOM}(\text{db}(s_2))$, such that $c_1 = c_2$, we have $\text{CERT}(Qv_1, T, \text{abox}(s_1)) = \text{ANS}(\text{rew}(Q, T)v_1, \text{abox}(s_1)) = \text{ANS}(\text{rew}(Q, T)v_2, \text{db}(s_2))$. Thus we have

$$\mathcal{T}_1, s_1 \models Qv_1 \text{ if and only if } \mathcal{T}_2, s_2 \models \text{rew}(Q, T)v_2$$

Inductive step:

- ($\Phi = \Psi_1 \vee \Psi_2$). we have $\mathcal{T}_1, s_1 \models (\Psi_1 \vee \Psi_2)v_1$ if and only if either $\mathcal{T}_1, s_1 \models \Psi_1v_1$ or $\mathcal{T}_1, s_1 \models \Psi_2v_1$. By induction hypothesis, for every (open) $\mu\mathcal{L}_A^{\text{EQL}}$ formula Ψ , and for every valuations v_1 and v_2 that assign to each of its free variables a constant $c_1 \in \text{ADOM}(\text{abox}(s_1))$ and $c_2 \in \text{ADOM}(\text{db}(s_2))$, such that $c_1 = c_2$, we have

- $\mathcal{T}_1, s_1 \models \Psi_1v_1$ if and only if $\mathcal{T}_2, s_2 \models \text{rew}(\Psi_1, T)v_2$, and also
- $\mathcal{T}_1, s_1 \models \Psi_2v_1$ if and only if $\mathcal{T}_2, s_2 \models \text{rew}(\Psi_2, T)v_2$

Hence, $\mathcal{T}_1, s_1 \models \Psi_1v_1$ or $\mathcal{T}_1, s_1 \models \Psi_2v_1$ if and only if $\mathcal{T}_2, s_2 \models \text{rew}(\Psi_1, T)v_2$ or $\mathcal{T}_2, s_2 \models \text{rew}(\Psi_2, T)v_2$. Therefore we have $\mathcal{T}_1, s_1 \models (\Psi_1 \vee \Psi_2)v_1$ if and only if $\mathcal{T}_2, s_2 \models (\text{rew}(\Psi_1, T) \vee \text{rew}(\Psi_2, T))v_2$. Since $\text{rew}(\Psi_1 \vee \Psi_2, T) = \text{rew}(\Psi_1, T) \vee \text{rew}(\Psi_2, T)$, we have

$$\mathcal{T}_1, s_1 \models (\Psi_1 \vee \Psi_2)v_1 \text{ if and only if } \mathcal{T}_2, s_2 \models \text{rew}(\Psi_1 \vee \Psi_2)v_2$$

($\Phi = \langle \neg \rangle \Psi$). Assume $\mathcal{T}_1, s_1 \models (\langle \neg \rangle \Psi)v_1$, where v_1 is a valuation that assigns to each free variable of Ψ a constant $c_1 \in \text{ADOM}(\text{abox}(s_1))$. Then there exists s'_1 s.t. $s_1 \Rightarrow_1 s'_1$ and $\mathcal{T}_1, s'_1 \models \Psi v_1$. Since $s_1 \sim_{\text{KD}} s_2$, there exists s'_2 such that $s_2 \Rightarrow_2 s'_2$ and $s'_1 \sim_{\text{KD}} s'_2$. Hence, by induction hypothesis, for every valuations v_2 that assign to each free variable x of $\text{rew}(\Psi, T)$ a constant $c_2 \in \text{ADOM}(\text{db}(s_2))$, such that $c_1 = c_2$, we have $\mathcal{T}_2, s'_2 \models \text{rew}(\Psi, T)v_2$. Consider that $s_2 \Rightarrow_2 s'_2$, we therefore get $\mathcal{T}_2, s_2 \models (\langle \neg \rangle \text{rew}(\Psi, T))v_2$. Since $\text{rew}(\langle \neg \rangle \Psi, T) = \langle \neg \rangle \text{rew}(\Psi, T)$, we have $\mathcal{T}_2, s_2 \models (\langle \neg \rangle \text{rew}(\Psi, T))v_2$. The other direction can be shown in a similar way.

($\Phi = \exists x. \Psi$). Assume that $\mathcal{T}_1, s_1 \models (\exists x. \Psi)v'_1$, where v'_1 is a valuation that assigns to each free variable of Ψ a constant $c_1 \in \text{ADOM}(\text{abox}(s_1))$. Then, by definition, there exists $c \in \text{ADOM}(\text{abox}(s_1))$ such that $\mathcal{T}_1, s_1 \models \Psi v_1$, where $v_1 = v'_1[x/c]$. By induction hypothesis, for every valuation v_2 that assigns to each free variable y of $\text{rew}(\Psi, T)$ a constant $c_2 \in \text{ADOM}(\text{db}(s_2))$, such that $c_2 = c_1$ with $y/c_1 \in v_1$, we have that $\mathcal{T}_2, s_2 \models \text{rew}(\Psi, T)v_2$. Additionally, $v_2 = v'_2[x/c']$, where $c' \in \text{ADOM}(\text{db}(s_2))$, and $c' = c$ because $\text{db}(s_2) = \text{abox}(s_1)$. Hence, we get $\mathcal{T}_2, s_2 \models (\exists x. \text{rew}(\Psi, T))v'_2$. Furthermore, since $\text{rew}(\exists x. \Psi, T) = (\exists x. \text{rew}(\Psi, T))$, we have $\mathcal{T}_2, s_2 \models (\text{rew}(\exists x. \Psi, T))v'_2$. The other direction can be shown similarly.

Extension to arbitrary countable disjunction. Let Ψ be a countable set of $\mathcal{L}_A^{\text{EQL}}$ formulae. Given either a KB transition system $\mathcal{T} = \langle \Delta, \mathcal{T}, \Sigma, s_0, \text{abox}, \Rightarrow \rangle$ (or a database transition system $\mathcal{T} = \langle \Delta, \mathcal{R}, \Sigma_2, s_{02}, \text{db}, \Rightarrow_2 \rangle$), the semantics of $\bigvee \Psi$ is $(\bigvee \Psi)_v^{\mathcal{T}} = \bigcup_{\psi \in \Psi} (\psi)_v^{\mathcal{T}}$. Therefore, given a state $s \in \Sigma$ we have $\mathcal{T}, s \models (\bigvee \Psi)v$ if and only if there exists $\psi \in \Psi$ such that $\mathcal{T}, s \models \psi v$. Arbitrary countable conjunction is obtained for free because of negation.

Now, let $\mathcal{T}_1 = \langle \Delta, \mathcal{T}, \Sigma_1, s_{01}, \text{abox}, \Rightarrow_1 \rangle$ and $\mathcal{T}_2 = \langle \Delta, \mathcal{R}, \Sigma_2, s_{02}, \text{db}, \Rightarrow_2 \rangle$. Consider two states $s_1 \in \Sigma_1$ and $s_2 \in \Sigma_2$ such that $s_1 \sim_{\text{KD}} s_2$. By induction hypothesis, we have for every valuations v_1 and v_2 that assign to each free variable of $\bigvee \Psi$ a constant $c_1 \in \text{ADOM}(\text{abox}(s_1))$ and $c_2 \in \text{ADOM}(\text{db}(s_2))$, such that $c_2 = c_1$, we have that for every formula $\psi \in \Psi$, it holds $\mathcal{T}_1, s_1 \models \psi v_1$ if and only if $\mathcal{T}_2, s_2 \models \text{rew}(\psi, T)v_2$. Given the semantics of $\bigvee \Psi$ above, this implies that $\mathcal{T}_1, s \models (\bigvee \Psi)v_1$ if and only if $\mathcal{T}_2, s \models (\bigvee \text{rew}(\Psi, T))v_2$, where $\text{rew}(\Psi, T) = \{\text{rew}(\psi, T) \mid \psi \in \Psi\}$. The proof is then obtained by observing that $\bigvee \text{rew}(\Psi, T) = \text{rew}(\bigvee \Psi, T)$.

Extension to full $\mu\mathcal{L}_A^{\text{EQL}}$. In order to extend the result to the whole $\mu\mathcal{L}_A^{\text{EQL}}$, we resort to the well-known result stating that fixpoints of the μ -calculus can be translated into the infinitary Hennessy Milner logic by iterating over *approximants*, where the approximant of index α is denoted by $\mu^\alpha Z. \Phi$ (resp. $\nu^\alpha Z. \Phi$) (cf. [170]). This is a standard result that also holds for $\mu\mathcal{L}_A^{\text{EQL}}$. In particular, approximants are built as follows:

$$\begin{aligned} \mu^0 Z. \Phi &= \text{false} & \nu^0 Z. \Phi &= \text{true} \\ \mu^{\beta+1} Z. \Phi &= \Phi[Z/\mu^\beta Z. \Phi] & \nu^{\beta+1} Z. \Phi &= \Phi[Z/\nu^\beta Z. \Phi] \\ \mu^\lambda Z. \Phi &= \bigvee_{\beta < \lambda} \mu^\beta Z. \Phi & \nu^\lambda Z. \Phi &= \bigwedge_{\beta < \lambda} \nu^\beta Z. \Phi \end{aligned}$$

where λ is a limit ordinal, and where fixpoints and their approximants are connected by the following properties: given a transition system \mathcal{T} and a state s of \mathcal{T}

- $s \in (\mu Z. \Phi)_{v,V}^{\mathcal{T}}$ if and only if there exists an ordinal α such that $s \in (\mu^\alpha Z. \Phi)_{v,V}^{\mathcal{T}}$ and, for every $\beta < \alpha$, it holds that $s \notin (\mu^\beta Z. \Phi)_{v,V}^{\mathcal{T}}$;

- $s \notin (\nu Z.\Phi)_{v,V}^r$ if and only if there exists an ordinal α such that $s \notin (\nu^\alpha Z.\Phi)_{v,V}^r$ and, for every $\beta < \alpha$, it holds that $s \in (\nu^\beta Z.\Phi)_{v,V}^r$.

□

Having Lemma 3.22 in hand, we can easily show the following important theorem which essentially says that given a KB transition system \mathcal{T}_1 that is KD-bisimilar to a database transition system \mathcal{T}_2 , we have that \mathcal{T}_1 satisfies a $\mu\mathcal{L}_A^{\text{EQL}}$ property Φ if and only if \mathcal{T}_2 satisfies a $\mu\mathcal{L}_A$ property Φ' that is obtained from Φ through perfect reformulation algorithm.

Theorem 3.23. *Consider a KB transition system $\mathcal{T}_1 = \langle \Delta, T, \Sigma_1, s_{01}, \text{abox}, \Rightarrow_1 \rangle$ and a database transition system \mathcal{T}_2 such that $\mathcal{T}_1 \sim_{\text{KD}} \mathcal{T}_2$. For every $\mu\mathcal{L}_A^{\text{EQL}}$ closed formula Φ , we have:*

$$\mathcal{T}_1 \models \Phi \text{ if and only if } \mathcal{T}_2 \models \text{rew}(\Phi, T)$$

Proof. Let $\mathcal{T}_2 = \langle \Delta, \mathcal{R}, \Sigma_2, s_{02}, \text{db}, \Rightarrow_2 \rangle$. Since $\mathcal{T}_1 \sim_{\text{KD}} \mathcal{T}_2$, then we have $s_{01} \sim_{\text{KD}} s_{02}$. Hence, by Lemma 3.22 we have

$$\mathcal{T}_1, s_{01} \models \Phi \text{ if and only if } \mathcal{T}_2, s_{02} \models \text{rew}(\Phi, T),$$

which prove the claim. □

Now we are going to show that the transition system of a KAB \mathcal{K} and the transition system of its corresponding DCDS $\tau_{\text{dcds}}(\mathcal{K})$ (obtained from \mathcal{K} via τ_{dcds}) are KD-bisimilar. As the first step, in Lemma 3.22, we show that given a state of a KAB transition system and a state of its corresponding DCDS transition system such that those two states contain the same data (i.e., their corresponding ABox and database instance are equal) as well as the same service call map, we have that they are KD-bisimilar.

Lemma 3.24. *Let \mathcal{K} be a KAB with transition system $\mathcal{T}_{\mathcal{K}}$, and let $\tau_{\text{dcds}}(\mathcal{K})$ be a DCDS obtained from \mathcal{K} through τ_{dcds} with transition system $\mathcal{T}_{\tau_{\text{dcds}}(\mathcal{K})}$. Consider a state $\langle A, m \rangle$ of $\mathcal{T}_{\mathcal{K}}$ and a state $\langle \mathbf{I}, m_d \rangle$ of $\mathcal{T}_{\tau_{\text{dcds}}(\mathcal{K})}$. If $A = \mathbf{I}$ and $m = m_d$, then $\langle A, m \rangle \sim_{\text{KD}} \langle \mathbf{I}, m_d \rangle$.*

Proof. Let

- $\mathcal{K} = \langle T, A_0, \Gamma, \Pi \rangle$, and $\mathcal{T}_{\mathcal{K}} = \langle \Delta, T, \Sigma, s_0, \text{abox}, \Rightarrow \rangle$,
- $\tau_{\text{dcds}}(\mathcal{K}) = \langle D, P \rangle$, where $D = \langle \mathcal{R}, \mathbf{I}_0, \mathcal{E} \rangle$, and $P = \langle \mathcal{A}, \varrho \rangle$. Additionally, let $\mathcal{T}_{\tau_{\text{dcds}}(\mathcal{K})} = \langle \Delta, \mathcal{R}, \Sigma, s_0, \text{db}, \Rightarrow \rangle$.

To prove the lemma, we show that for every state $\langle A', m' \rangle$ such that $\langle A, m \rangle \Rightarrow \langle A', m' \rangle$, there exists a state $\langle \mathbf{I}', m'_d \rangle$ such that

1. $\langle \mathbf{I}, m_d \rangle \Rightarrow \langle \mathbf{I}', m'_d \rangle$
2. $A' = \mathbf{I}'$
3. $m' = m'_d$

By definition of $\mathcal{T}_{\mathcal{K}}$ (Definition 3.11), since $\langle A, m \rangle \Rightarrow \langle A', m' \rangle$, then there exists $\alpha \in \Gamma$, and a substitution σ for parameters of α such that $\langle A, m \rangle \xrightarrow{\alpha\sigma, \mathcal{K}} \langle A', m' \rangle$ and A' is T -consistent. Moreover, by definition of $\xrightarrow{\alpha\sigma, \mathcal{K}}$ (see Definition 3.10), we have the following:

1. σ is a legal parameter assignment for α in state $\langle A, m \rangle$, and additionally by Definition 3.5, there exists a condition-action rule $Q \mapsto \alpha \in \Pi$ such that $\text{CERT}(Q\sigma, T, A)$ is true;
2. there exists $\theta \in \text{EVAL}(T, A, \alpha\sigma)$ such that θ and m “agree” on the common values in their domains;
3. $A' = \text{DO}(T, A, \alpha\sigma)\theta$; and
4. $m' = m \cup \theta$ (i.e., updating the history of issued service calls).

By definition of τ_{dcds} (Definition 3.16) we have the following:

- there exists a corresponding action $\alpha' \in \mathcal{A}$ which is obtained from $\alpha \in \Gamma$,
- there exists a corresponding condition-action rule $Q' \mapsto \alpha' \in \varrho$ (where $Q' = \text{rew}(Q, T)$) which is obtained from $Q \mapsto \alpha \in \Pi$

Since $A = \mathbf{I}$, by Theorem 2.40, we have

$$\text{CERT}(Q, T, A) = \text{ANS}(\text{rew}(Q, T), A) = \text{ANS}(Q', \mathbf{I}).$$

Hence, by Definition 2.56, σ is also a legal parameter assignment for α' in state $\langle \mathbf{I}, m_d \rangle$. Therefore, by Lemma 3.17, we have $\text{DO}(T, A, \alpha\sigma) = \text{DO}_{\text{DCDS}}(\mathbf{I}, \alpha'\sigma)$. Thus, by Lemma 3.18, we have $\text{EVAL}(T, A, \alpha\sigma) = \text{EVAL}(\mathbf{I}, \alpha'\sigma)$, and hence we have $\theta \in \text{EVAL}(\mathbf{I}, \alpha'\sigma)$. As a consequence, we have

1. $A' = \text{DO}(T, A, \alpha\sigma)\theta = \text{DO}_{\text{DCDS}}(\mathbf{I}, \alpha'\sigma)\theta = \mathbf{I}'$.
2. $m' = m'_d$, because $m_d = m$, $m' = m \cup \theta$, and $m'_d = m_d \cup \theta$.

Furthermore, it is easy to see that we have $\langle \mathbf{I}, m \rangle \xrightarrow{\alpha'\sigma, \mathcal{S}} \langle \mathbf{I}', m' \rangle$, and since $A' = \mathbf{I}'$ as well as A' is T -consistent, by Lemma 3.19 we have \mathbf{I}' satisfies \mathcal{E} . Hence by Definition 2.60, we have $\langle \mathbf{I}, m_d \rangle \Rightarrow \langle \mathbf{I}', m'_d \rangle$.

The other direction of bisimulation can be shown similarly. \square

Using Lemma 3.22 above, now we show that the transition system of a KAB \mathcal{K} is KD-bisimilar with the transition system of the corresponding DCDS $\tau_{dcds}(\mathcal{K})$ (that is obtained from \mathcal{K} via τ_{dcds}) as follows.

Theorem 3.25. *Given a KAB \mathcal{K} with transition system $\Upsilon_{\mathcal{K}}$, and let $\tau_{dcds}(\mathcal{K})$ be a DCDS obtained from \mathcal{K} through τ_{dcds} with transition system $\Upsilon_{\tau_{dcds}(\mathcal{K})}$. We have $\Upsilon_{\mathcal{K}} \sim_{\text{KD}} \Upsilon_{\tau_{dcds}(\mathcal{K})}$*

Proof. Let

- $\mathcal{K} = \langle T, A_0, \Gamma, \Pi \rangle$, and $\Upsilon_{\mathcal{K}} = \langle \Delta, T, \Sigma, s_{0k}, \text{abox}, \Rightarrow \rangle$,
- $\tau_{dcds}(\mathcal{K}) = \langle D, P \rangle$, where $D = \langle \mathcal{R}, \mathbf{I}_0, \mathcal{E} \rangle$, and $P = \langle \mathcal{A}, \varrho \rangle$. Additionally, let $\Upsilon_{\tau_{dcds}(\mathcal{K})} = \langle \Delta, \mathcal{R}, \Sigma, s_{0d}, \text{db}, \Rightarrow \rangle$.

By Definition 3.11, we have $s_{0k} = \langle A_0, \emptyset \rangle$, and by Definition 2.60, we have $s_{0d} = \langle \mathbf{I}_0, \emptyset \rangle$. Furthermore, by the definition of τ_{dcds} (Definition 3.16), we have $A_0 = \mathbf{I}_0$. Hence by Lemma 3.22, we have $s_{0k} \sim_{\text{KD}} s_{0d}$. Thus, we have $\Upsilon_{\mathcal{K}} \sim_{\text{KD}} \Upsilon_{\tau_{dcds}(\mathcal{K})}$ \square

Having the fact that the transition system of a KAB \mathcal{K} is KD-bisimilar with the transition system of the corresponding DCDS obtained from \mathcal{K} via translation τ_{dcds} , by exploiting Theorem 3.23 we can finally show that the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ properties over KABs can be reduced to the verification of $\mu\mathcal{L}_A$ properties over DCDSs as follows.

Theorem 3.26. *Given a KAB \mathcal{K} and a closed $\mu\mathcal{L}_A^{\text{EQL}}$ formula Φ , we have*

$$\Upsilon_{\mathcal{K}} \models \Phi \quad \text{if and only if} \quad \Upsilon_{\tau_{dcds}(\mathcal{K})} \models \text{rew}(\Phi, T)$$

Proof. By Theorem 3.25, we have $\mathcal{T}_{\mathcal{K}} \sim_{\text{KD}} \mathcal{T}_{\tau_{dc ds}(\mathcal{K})}$. Hence, the claim is directly follows from Theorem 3.23. \square

Finally, Theorem 3.26 shows that we can tackle the problem of $\mu\mathcal{L}_A^{\text{EQL}}$ verification over KABs by reducing it into the problem of $\mu\mathcal{L}_A$ verification over DCDSs.

3.3.4 Verification of Run-Bounded KABs

As in DCDSs, in general the verification of KABs is undecidable. However, we can use the semantic restriction that was originally proposed in [24], namely *run-boundedness* in order to gain decidability. To introduce the notion of run-boundedness, we first define the notion of run of a KAB transition system as follows.

Definition 3.27 (Run of a KAB Transition System). Given a KAB \mathcal{K} , a *run of* $\mathcal{T}_{\mathcal{K}} = \langle \Delta, T, \Sigma, s_0, \text{abox}, \Rightarrow \rangle$ is a (possibly infinite) sequence $s_0 s_1 \dots$ of states of $\mathcal{T}_{\mathcal{K}}$ such that $s_i \Rightarrow s_{i+1}$, for all $i \geq 0$. *Run of a KAB Transition System* \blacksquare

Definition 3.28 (Run-bounded KAB). Given a KAB \mathcal{K} , we say \mathcal{K} is *run-bounded* if there exists an integer bound b such that for every run $\pi = s_0 s_1 \dots$ of $\mathcal{T}_{\mathcal{K}}$, we have that $|\bigcup_{s \text{ state of } \pi} \text{ADOM}(\text{abox}(s))| < b$. *Run-bounded KAB* \blacksquare

Intuitively, run-boundedness requires that every run in the transition system cumulatively encounters at most a bounded number of constants. Unboundedly many constants can still be present in the overall system, provided that they do not accumulate in the same run.

Lemma 3.29. *Given a run-bounded KAB \mathcal{K} , the DCDS $\tau_{dc ds}(\mathcal{K})$ is run-bounded.*

Proof. Let

- $\mathcal{K} = \langle T, A_0, \Gamma, \Pi \rangle$, and $\mathcal{T}_{\mathcal{K}} = \langle \Delta, T, \Sigma, s_0^k, \text{abox}, \Rightarrow \rangle$ be its corresponding transition system,
- $\tau_{dc ds}(\mathcal{K}) = \langle D, P \rangle$, and $\mathcal{T}_{\tau_{dc ds}(\mathcal{K})} = \langle \Delta, \mathcal{R}, \Sigma, s_0^d, db, \Rightarrow \rangle$ be its corresponding transition system,
- $\pi_k = s_0^k s_1^k \dots$ be an arbitrary run of $\mathcal{T}_{\mathcal{K}}$.

Since, \mathcal{K} is run-bounded, we have that there exists an integer bound b such that $|\bigcup_{s^k \text{ state of } \pi_k} \text{ADOM}(\text{abox}(s^k))| < b$. By Theorem 3.25 we have that $\mathcal{T}_{\mathcal{K}} \sim_{\text{KD}} \mathcal{T}_{\tau_{dc ds}(\mathcal{K})}$. As a consequence, there exists a corresponding run $\pi_d = s_0^d s_1^d \dots$ in $\mathcal{T}_{\tau_{dc ds}(\mathcal{K})}$ such that $\text{abox}(s_i^k) = db(s_i^d)$ (for $i = 0, 1, \dots$). Thus we get that there exists an integer bound b such that $|\bigcup_{s^d \text{ state of } \pi_d} \text{ADOM}(db(s^d))| < b$. The proof is then completed by noticing that

1. π_k is an arbitrary run of $\mathcal{T}_{\mathcal{K}}$, and
2. because $\mathcal{T}_{\mathcal{K}} \sim_{\text{KD}} \mathcal{T}_{\tau_{dc ds}(\mathcal{K})}$, for each run $\pi_d = s_0^d s_1^d \dots$ in $\mathcal{T}_{\tau_{dc ds}(\mathcal{K})}$ there exists a corresponding run $\pi_k = s_0^k s_1^k \dots$ in $\mathcal{T}_{\mathcal{K}}$ such that $\text{abox}(s_i^k) = db(s_i^d)$ (for $i = 0, 1, \dots$).

\square

Finally, we can state the final result on verification of $\mu\mathcal{L}_A^{\text{EQL}}$ over run-bounded KAB.

Theorem 3.30 (Verification of $\mu\mathcal{L}_A^{\text{EQL}}$ over run-bounded KAB). *Verification of closed $\mu\mathcal{L}_A^{\text{EQL}}$ formulas over run-bounded KAB is decidable and can be reduced to finite-state model checking.*

Proof. From Theorem 3.26 and Lemma 3.29, we have that verification of closed $\mu\mathcal{L}_A^{\text{EQL}}$ formulas over run-bounded KAB can be reduced to the verification of $\mu\mathcal{L}_A$ formulas over run-bounded DCDS. Then, by Theorem 2.65, we have that verification of $\mu\mathcal{L}_A$ over run-bounded DCDS is decidable and can be reduced to finite-state model checking. \square

3.4 Discussion: Weakly Acyclic KABs

Studying various kind of restrictions to obtain decidability is out of the scope of this thesis. However, in the following we provide some discussions on the condition for obtaining decidability of verification, in particular on the notion of *weak-acyclicity* that is a syntactic condition which implies (guarantees) run-boundedness.

As we have seen above, to get decidability of verification, in this thesis we rely on the assumption that the considered KAB is run-bounded. In [24], a sufficient syntactic condition borrowed from *weak acyclicity* in data exchange [100] has been studied. Such condition is shown to guarantee run boundedness under the assumption that the service calls are deterministic.

We can recast such notion of weak acyclicity that was studied in DCDSs into KABs such that if we have a KAB \mathcal{K} is weak acyclic, then \mathcal{K} is run bounded. Intuitively, given a KAB \mathcal{K} , this weak acyclicity test constructs a *dependency graph* tracking how the actions of \mathcal{K} transport values from one state to the next one. To track all the actual dependencies, every involved query is first rewritten considering the positive inclusion assertions of the TBox. Two types of dependencies are tracked:

1. copy of values and
2. use of values as parameters of a service call.

The KAB \mathcal{K} is said to be *weakly acyclic* if there is no cyclic chain of dependencies of the second kind. Intuitively, the presence of such a cycle could produce an infinite chain of fresh values generation through service calls. Thus, such a cycle could destroy run-boundedness. In other word, a non-weakly acyclic KAB contains at least a service that might be called repeatedly, and each call is using fresh values that are either directly or indirectly obtained by manipulating the previous result that is produced by the same service. Note that this notion of weak acyclicity is the same as the one in [121], that is used by [121] to get decidability of verification.

GOLOG-KABs (GKABs)

Knowledge and Action Bases (KABs) have been put forward as a framework which provides a semantically rich representation of a domain that also simultaneously takes into account the dynamic aspects of the modeled system. However, KABs lack of a convenient way to specify processes at a high-level of abstraction. To cope with this situation, we enrich KABs with a high-level, compact action language inspired by Golog [134]. We call Golog-KAB (GKAB) the KAB enhanced with a Golog-like programming language. Additionally, here we also introduce a parametric execution semantics for GKABs, so as to elegantly accomodate various way of updating an ABox. We will see later in the next chapter that the parametric execution semantics allow us to incorporate various inconsistency-aware execution semantics for GKABs.

In the following, we use $DL-Lite_A$ for expressing knowledge bases and we also do not distinguish between objects and values (thus we drop attributes). Furthermore, we also make use of a countably infinite set Δ of constants, which intuitively denotes all possible values in the system. Additionally, we consider a finite set of distinguished constants $\Delta_0 \subset \Delta$, and a finite set \mathcal{F} of *function symbols* that represents *service calls*, which abstractly account for the injection of fresh values (constants) from Δ into the system. The results in this chapter are published in [75, 77, 76].

4.1 GKABs Formalism

In this section, we enrich KABs (cf. Chapter 3) with a high-level action language inspired by Golog [134]. This allows modelers to represent processes in a much more intuitive and compact way. In this thesis, we consider a variant of Golog that has been tailored to work on KBs based on [55]. Formally, a *Golog program* is inductively defined as follows:

Definition 4.1 (Golog Program). Given a set of KAB actions Γ (see Definition 3.1), a *Golog program* δ over Γ is an expression formed by the following grammar:

$$\begin{aligned} \delta ::= & \varepsilon \mid \mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p}) \mid \delta_1 \mid \delta_2 \mid \delta_1; \delta_2 \mid \\ & \mathbf{if} \ \varphi \ \mathbf{then} \ \delta_1 \ \mathbf{else} \ \delta_2 \mid \mathbf{while} \ \varphi \ \mathbf{do} \ \delta \end{aligned}$$

where:

- ε is the *empty program*;
- $\mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p})$ is an *atomic action invocation* guarded by a DI-ECQ Q , such that $\alpha \in \Gamma$ is applied by non-deterministically substituting its parameters \vec{p} with an answer of Q . Additionally, $Q(\vec{p})$ might uses constants in Δ_0 ;
- $\delta_1 \mid \delta_2$ is a *non-deterministic choice* between programs;
- $\delta_1; \delta_2$ is *sequencing*;
- $\mathbf{if} \ \varphi \ \mathbf{then} \ \delta_1 \ \mathbf{else} \ \delta_2$ and $\mathbf{while} \ \varphi \ \mathbf{do} \ \delta$ are *conditional* and *loop* constructs, using a boolean ECQ φ as condition.

■

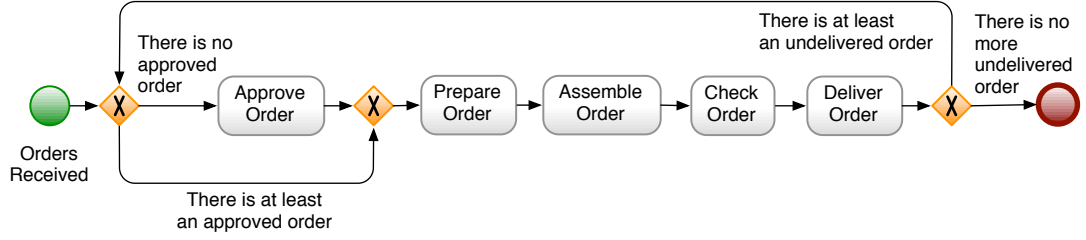


Figure 7: Simple Order Processing Scenario in a Furniture Company

Notice that we are able to simulate some of other Golog program constructs by using the constructs above. We discuss such possibilities in Section 4.5.

We then define the notion of Golog-KABs as follows.

Golog-KAB **Definition 4.2** (Golog-KAB). A *Golog-KAB* (*GKAB*) is a tuple $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$, where

- T and A_0 together form a satisfiable *DL-Lite_A* KB $\langle T, A_0 \rangle$, where
 - T is a *DL-Lite_A* TBox that captures the intensional aspects of the domain of interest;
 - A_0 is the *initial DL-Lite_A* ABox, describing the initial configuration of data;
- Γ is a finite set of *KAB actions* (as in Definition 3.1) over $\langle T, A_0 \rangle$ that evolve the ABox;
- δ is a Golog program over Γ , which characterizes the evolution of the GKAB over time, using the atomic actions in Γ .

Additionally, we assume that $\text{ADOM}(A_0) \subseteq \Delta_0$. ■

The crucial difference between a GKAB and a KAB is that a GKAB specifies its processes using a Golog program instead of a set of condition-action rules.

Example 4.3. An example of a GKAB.

Consider the furniture company order processing scenario described in Section 2.1. For the running example in this chapter, we slightly adjust the scenario as follows:

1. The order processing flows are followed strictly in a sequential manner (e.g., order preparation must be followed by order assembling).
2. In the beginning of the processing flow, in case there is already at least an approved order, the company immediately starts to process the order. Otherwise, the company must first approve an order.
3. While there is still an order that is not yet delivered, the processes will be repeated.

The scenario that we consider here is visually described in Figure 7. We then model this scenario by a GKAB $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ where T , A_0 , Γ , and δ are specified as follows.

To capture the domain knowledge within this scenario, we consider the TBox T that is specified in Example 2.17. As for the initial ABox, our GKAB \mathcal{G} has the following initial ABox:

$$A_0 = \{\text{ReceivedOrder}(\text{chair}), \text{ApprovedOrder}(\text{table})\}$$

that basically contains a fact that there is an order of chair and a fact that there is an approved order of table.

To model the progression mechanism in the scenario above, we consider the set Γ of actions that is specified in Example 3.4. The order processing flow in the scenario above is then captured by the program δ of our GKAB \mathcal{G} , and it is specified as follows:

$$\delta = \textbf{while } \exists x. [\text{Order}(x)] \wedge \neg [\text{DeliveredOrder}(x)] \textbf{ do } \delta_0$$

where:

- $\delta_0 = \delta_1; \delta_2; \delta_3; \delta_4; \delta_5$,
- $\delta_1 = \textbf{if } \neg [\exists x. \text{ApprovedOrder}(x)]$
 then pick $[\text{ReceivedOrder}(x)].\text{approveOrder}(x)$
 else ε ,
- $\delta_2 = \textbf{pick true.prepareOrders}()$,
- $\delta_3 = \textbf{pick true.assembleOrders}()$,
- $\delta_4 = \textbf{pick true.checkAssembledOrders}()$,
- $\delta_5 = \textbf{pick true.deliverOrders}()$.

The intuition of the program δ above is as follows:

- **while** $\exists x. [\text{Order}(x)] \wedge \neg [\text{DeliveredOrder}(x)]$ **do** δ_0 states the fact that as long as there exists an order that is not yet delivered, the program δ_0 , which basically processes the orders, will be executed.
- δ_0 specifies a consecutive sequence of programs $(\delta_1; \delta_2; \delta_3; \delta_4; \delta_5)$ where each of them captures a certain activity related to the order processing.
- In δ_1 , if there does not exist any approved order, the program will pick up an order and then approve it by executing the action **approveOrder**/1 otherwise it will perform nothing.
- The programs $\delta_2, \delta_3, \delta_4, \delta_5$ consecutively perform the order preparation, order assembling, order quality control, and order delivery. Each of them is performed by consecutively executing the actions **prepareOrders**/0, **assembleOrders**/0, **checkAssembledOrders**/0, and **deliverOrders**/0.

With a slightly abuse of notation, in order to ease the understanding, below we provide another way to write the program above (note: we use the curly braces (“{”, “}”) to mark the scope of program constructor):

```

while  $\exists x. [\text{Order}(x)] \wedge \neg [\text{DeliveredOrder}(x)]$  do {
  if  $\neg [\exists x. \text{ApprovedOrder}(x)]$ 
    then {pick  $[\text{ReceivedOrder}(x)].\text{approveOrder}(x)$ }
    else { $\varepsilon$ };
  pick true.prepareOrders();
  pick true.assembleOrders();
  pick true.checkAssembledOrders();
  pick true.deliverOrders()
}

```

4.2 GKABs Standard Execution Semantics

Similar to standard KABs, the execution semantics of a GKAB $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ is given in terms of a possibly infinite-state KB transition system $\Upsilon_{\mathcal{G}} = \langle \Delta, T, \Sigma, s_0, abox, \Rightarrow \rangle$, whose states contain ABoxes (see Definition 2.45 for the detail of $\Upsilon_{\mathcal{G}}$ components). However, differently from KABs, the states we consider are tuples of the form $\langle A, m, \delta \rangle$, where A is an ABox, m a service call map (see Definition 2.55), and δ a program. Together, A and m constitute the *data-state*, which captures the result of the actions executed so far, together with the answers returned by service calls issued in the past. Instead, δ is the *program-state*, which represents the program that still needs to be executed from the current data-state. Later on, a state of the form $\langle A, m, \delta \rangle$ is often also called a *GKAB state*.

We adopt the functional approach by [135] in defining the semantics of action execution over GKAB \mathcal{G} , i.e., we assume \mathcal{G} provides two operations:

1. ASK, to answer queries over the current KB;
2. TELL, to update the KB through an atomic action.

By adopting the functional approach, we are able to interact with the KB as a black box through some operations (ASK/TELL). As a result, we can separate the reasoning over the structural/static aspect from the reasoning over the dynamic aspect while still capturing the combined behavior of both aspects within a system. As a consequence, it enables us to take advantage from their existing technique for the two kinds of reasoning (i.e., reasoning over DL KBs and reasoning over the evolution of the system that is characterized by action execution). Furthermore, note that KABs [121], which is the underlying framework of GKABs, is also already adopting functional approach in defining their semantics. As pointed out by [121], combining incomplete information setting (DL KB) with the capability to capture the dynamics of the system in one setting often brought us into a rich setting that is fragile w.r.t. undecidability aspects (cf. [183]). As it has been studied in the area of temporal description logics [138, 8], that combine both temporal logics and description logics, one crucial factor that influence the computational complexity of such setting is the degree of interaction between the temporal component and the DL component [138]. For instance, the work in [17, 18] obtain a nice decidability result for the combination of the Temporal Logic LTL [27] and the Description Logic \mathcal{ALC} [16] by limiting the interaction of those two components compare to the works in [10, 9] which either restrict the expressivity of the DL or the expressivity of the temporal component. In particular, [17, 18] only allows the temporal operators to be occurred in front of TBox/ABox assertions instead of in front of any concepts/roles. In KABs, to overcome such difficulties and obtain a robust result, the work of [121] follows the functional approach which gives a good control over the interaction between the DL KB and the dynamic aspects. Similar arguments also pointed out by [72]. By adopting the functional approach, [121] obtained a setting that is not capturing the dynamics/evolution of each single model of a DL KB, but it captures the evolution of KBs in which at each single step of evolution we consider only the portion of knowledge that hold in all possible models of the KB (i.e., the certain answers). As a consequence, this setting is typically computationally easier to handle.

Still about functional approach to knowledge representation, from the business processes point of view (in particular data-aware business processes), the motivation

is to capture the manipulation/evolution of data by the processes. Furthermore, each single operation (that manipulates the data) views a data storage as a black box in which they can manipulate the data inside it (i.e., by retrieving/querying the data as well as adding/deleting the data). After the execution of such operation, as a result, we have the data storage in which its data has been manipulated (cf. [93, 125, 85, 21]). Now, when we want to add the intensional domain knowledge into our data-aware business processes systems, the information storage is modeled by a KB that contains not only data but also the intensional domain knowledge. Hence, by adopting the functional approach, we can obtain a setting where each manipulation operation views the KB as a black box and they can simply perform some “direct” manipulation, i.e., retrieving (“asking”) some objects/data from the KB as well as “telling” how the KB should change. Moreover, as a result of the execution of a manipulation operation, we have the manipulated KB. Overall, we obtain a setting that captures the manipulation of KB by processes.

In this thesis, we consider that the ASK operator corresponds to certain answers computation of queries.

Definition 4.4 (ASK Operation). Given a KB $\langle T, A \rangle$, and a UCQ q (resp. an ECQ Q), we define $\text{ASK}(q, T, A) = \text{cert}(q, T, A)$ (resp. $\text{ASK}(Q, T, A) = \text{CERT}(Q, T, A)$). ■ ASK Operation

We proceed now to formally define TELL. As the first step, we introduce several preliminaries as follows.

Definition 4.5 (Executability of an Action Invocation). Let $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ be a GKAB. Given an action invocation **pick** $Q(\vec{p}).\alpha(\vec{p})$ and an ABox A , we say that a substitutions σ , which substitutes the parameters \vec{p} with constants in Δ , is a *legal parameter assignment for α in A w.r.t. **pick** $Q(\vec{p}).\alpha(\vec{p})$* if $\text{ASK}(Q\sigma, T, A)$ is true. In this case we also say that α is *executable in A with a legal parameter assignment σ* ■ Executability of an Action Invocation

Example 4.6. Recall our running example in Example 4.3, let $A = \{\text{ReceivedOrder}(\text{chair})\}$ be an ABox and consider the action invocation **pick** $[\text{ReceivedOrder}(x)].\text{approveOrder}(x)$. In this case the substitution σ that substitute x with chair (i.e., $\sigma(x) = \text{chair}$) is a legal parameter assignment for approveOrder in A w.r.t. **pick** $[\text{ReceivedOrder}(x)].\text{approveOrder}(x)$ since $\text{ASK}([\text{ReceivedOrder}(x)]\sigma, T, A)$ is true. Thus, approveOrder is executable in A with a legal parameter assignment σ .

With a slight abuse of the definition, we sometimes say that an action α is *executable in a state s with a legal parameter assignment σ* if $s = \langle A, m \rangle$ and α is executable in A with a legal parameter assignment σ . The notion of grounded action $\alpha\sigma$ is defined similarly as in KABs (i.e., a grounded action $\alpha\sigma$ is obtained by applying σ to each $e \in \text{EFF}(\alpha)$).

We define the sets $\text{ADD}(T, A, \alpha\sigma)$ (resp. $\text{DEL}(T, A, \alpha\sigma)$) of atoms to be added and deleted by **pick** $Q(\vec{p}).\alpha(\vec{p})$ w.r.t. σ in A similar to the definitions in KABs (see Definitions 3.7 and 3.8) as follows:

*Set of Atoms to be
Added By an Action*

Definition 4.7 (Set of Atoms to be Added). Let $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ be a GKAB. Given an ABox A , an action invocation **pick** $Q(\vec{p}).\alpha(\vec{p})$ where $\alpha \in \Gamma$ is an action of the form $\alpha(\vec{p}) : \{e_1, \dots, e_m\}$ with $e_i = [q_i^+] \wedge Q_i^- \rightsquigarrow \mathbf{add} F_i^+, \mathbf{del} F_i^-$, and a legal parameter assignment σ for α in A w.r.t. **pick** $Q(\vec{p}).\alpha(\vec{p})$, we define a *set of atoms to be added by* $\alpha\sigma$ w.r.t A as follows:

$$\text{ADD}(T, A, \alpha\sigma) = \bigcup_{([q^+] \wedge Q^- \rightsquigarrow \mathbf{add} F^+, \mathbf{del} F^-) \text{ in } \text{EFF}(\alpha)} \bigcup_{\rho \in \text{ASK}([q^+] \wedge Q^-)\sigma, T, A)} F^+ \sigma \rho$$

■

*Set of Atoms to be
Deleted By an Action*

Definition 4.8 (Set of Atoms to be Deleted). Let $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ be a GKAB. Given an ABox A , an action invocation **pick** $Q(\vec{p}).\alpha(\vec{p})$ where $\alpha \in \Gamma$ is an action of the form $\alpha(\vec{p}) : \{e_1, \dots, e_m\}$ with $e_i = [q_i^+] \wedge Q_i^- \rightsquigarrow \mathbf{add} F_i^+, \mathbf{del} F_i^-$, and a legal parameter assignment σ for α in A w.r.t. **pick** $Q(\vec{p}).\alpha(\vec{p})$, we define a *set of atoms to be deleted by* $\alpha\sigma$ w.r.t A as follows:

$$\text{DEL}(T, A, \alpha\sigma) = \bigcup_{([q^+] \wedge Q^- \rightsquigarrow \mathbf{add} F^+, \mathbf{del} F^-) \text{ in } \text{EFF}(\alpha)} \bigcup_{\rho \in \text{ASK}([q^+] \wedge Q^-)\sigma, T, A)} F^- \sigma \rho$$

■

Example 4.9. Consider our running example in Example 4.3. Let

$$A = \{\text{ReceivedOrder}(\text{chair}), \text{ApprovedOrder}(\text{table})\}$$

be an ABox. Consider the action invocation **pick** `true.assembleOrders()` and a legal parameter assignment σ for the action `assembleOrders` in A (in this case σ is an empty substitution because the atomic invocation **pick** `true.assembleOrders()` is guarded by `true`). Then we have that

- the set $\text{ADD}(T, A, \text{assembleOrders}\sigma)$ of atoms to be added by `assembleOrders` σ is

$$\{ \text{AssembledOrder}(\text{table}), \text{assembledBy}(\text{table}, \text{GETASSEMBLER}(\text{table})), \\ \text{Assembler}(\text{GETASSEMBLER}(\text{table})), \\ \text{hasAssemblingLoc}(\text{table}, \text{GETASSEMBLINGLOC}(\text{table})) \},$$

- the set $\text{DEL}(T, A, \text{assembleOrders}\sigma)$ of atoms to be deleted by `assembleOrders` σ is $\{\text{ApprovedOrder}(\text{table})\}$.

*Service Call
Evaluation*

In general, $\text{ADD}(T, A, \alpha\sigma)$ is not a proper set of ABox assertions, because it could contain (ground) skolem terms (representing service calls), to be substituted with corresponding results. We denote by $\text{CALLS}(\text{ADD}(T, A, \alpha\sigma))$ the set of such ground skolem terms in $\text{ADD}(T, A, \alpha\sigma)$, and by $\text{EVAL}(\text{ADD}(T, A, \alpha\sigma))$ the set of substitutions that replace such skolem terms with concrete values (constants) in Δ . Specifically, $\text{EVAL}(\text{ADD}(T, A, \alpha\sigma))$ is defined as follows:

$$\text{EVAL}(\text{ADD}(T, A, \alpha\sigma)) = \{ \theta \mid \theta \text{ is a total function,} \\ \theta : \text{CALLS}(\text{ADD}(T, A, \alpha\sigma)) \rightarrow \Delta \},$$

and $\text{EVAL}(\text{ADD}(T, A, \alpha\sigma))$ is a singleton set of an empty substitution θ when $\text{CALLS}(\text{ADD}(T, A, \alpha\sigma)) = \emptyset$.

Example 4.10. Continuing Example 4.9, we have that

$$\begin{aligned} \text{CALLS}(\text{ADD}(T, A, \text{assembleOrders}\sigma)) = \\ \{\text{GETASSEMBLER}(\text{table}), \text{GETASSEMBLINGLOC}(\text{table})\}. \end{aligned}$$

and $\text{EVAL}(\text{ADD}(T, A, \text{assembleOrders}\sigma))$ contains a set of substitutions that substitute the ground skolem terms $\text{GETASSEMBLER}(\text{table})$ and $\text{GETASSEMBLINGLOC}(\text{table})$ into constants in Δ . For example, a substitution θ where

1. $\theta(\text{GETASSEMBLER}(\text{table})) = \text{john}$,
2. $\theta(\text{GETASSEMBLINGLOC}(\text{table})) = \text{bolzano}$, and
3. $\{\text{john}, \text{bolzano}\} \subseteq \Delta$,

is in $\text{EVAL}(\text{ADD}(T, A, \text{assembleOrders}\sigma))$.

As the last step towards defining the TELL relations, given two ABoxes A and A' where A is assumed to be T -consistent, and two sets F^+ and F^- of ABox assertions, we introduce so-called *filter relation* to indicate that A' is obtained from A by adding the ABox assertions in F^+ and removing the one in F^- as follows.

Definition 4.11 (Filter Relation). Given a T -consistent ABox A , a *filter relation* f is a relation that consists of tuples of the form $\langle A, F^+, F^-, A' \rangle$ such that we have $\emptyset \subseteq A' \subseteq ((A \setminus F^-) \cup F^+)$, where A and A' are ABoxes, and F^+ as well as F^- are two sets of ABox assertions. ■

Filter Relation

In this light, filter relations provide an abstract mechanism to accommodate various approaches in updating an ABox. For example, to account for inconsistencies, the filter could drop some additional facts when producing A' .

Having all ingredients in hand, we are now ready to define the TELL operation as follows.

Definition 4.12 (TELL Operation). Given a GKAB $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ and a filter f , we define TELL_f as a relation over pairs of data-states such that we have a tuple $\langle \langle A, m \rangle, \alpha\sigma, \langle A', m' \rangle \rangle \in \text{TELL}_f$ if

TELL Operation

- σ is a legal parameter assignment for α in A w.r.t. a certain action invocation **pick** $Q(\vec{p}).\alpha(\vec{p})$, and
- there exists $\theta \in \text{EVAL}(\text{ADD}(T, A, \alpha\sigma))$ such that:
 1. for each skolem term $f(c) \in \text{DOM}(m) \cap \text{DOM}(\theta)$, we have $f(c)/v \in m$ if and only if $f(c)/v \in \theta$ (i.e., θ and m “agree” on the common skolem terms in their domains, in order to realize the deterministic service call semantics);
 2. $m' = m \cup \theta$;
 3. $\langle A, \text{ADD}(T, A, \alpha\sigma)\theta, \text{DEL}(T, A, \alpha\sigma), A' \rangle \in f$, where $\text{ADD}(T, A, \alpha\sigma)\theta$ denotes the set of ABox assertions obtained by applying θ over each element of $\text{ADD}(T, A, \alpha\sigma)$;
 4. A and A' is T -consistent.

■

Example 4.13. Continuing Example 4.10, let $m = \emptyset$. Suppose that we have

$$\langle A, \text{ADD}(T, A, \text{assembleOrders}\sigma)\theta, \text{DEL}(T, A, \text{assembleOrders}\sigma), A' \rangle \in f$$

where

1. $\theta(\text{GETASSEMBLER}(\text{table})) = \text{john}$,
2. $\theta(\text{GETASSEMBLINGLOC}(\text{table})) = \text{bolzano}$, and
3. $\{\text{john}, \text{bolzano}\} \subseteq \Delta$,
4. $\text{ADD}(T, A, \text{assembleOrders}\sigma)\theta = \left\{ \begin{array}{l} \text{AssembledOrder}(\text{table}), \\ \text{assembledBy}(\text{table}, \text{john}), \\ \text{Assembler}(\text{john}), \\ \text{hasAssemblingLoc}(\text{table}, \text{bolzano}) \end{array} \right\}$
5. A' is as follows:

$$\begin{aligned} A' &= (A \setminus \text{DEL}(T, A, \text{assembleOrders}\sigma)) \cup \text{ADD}(T, A, \text{assembleOrders}\sigma)\theta \\ &= \{ \text{ReceivedOrder}(\text{chair}), \text{AssembledOrder}(\text{table}), \\ &\quad \text{assembledBy}(\text{table}, \text{john}), \text{hasAssemblingLoc}(\text{table}, \text{bolzano}), \\ &\quad \text{Assembler}(\text{john}) \}. \end{aligned}$$

Then we have $\langle \langle A, m \rangle, \text{assembleOrders}\sigma, \langle A', m' \rangle \rangle \in \text{TELL}_f$ where $m' = m \cup \theta$. Intuitively, $\langle A', m' \rangle$ represents the result of execution of $\text{assembleOrders}\sigma$ over $\langle A, m \rangle$ where A' is obtained from A by first deleting the facts to be deleted by $\text{assembleOrders}\sigma$ and then add the facts to be added by $\text{assembleOrders}\sigma$.

As the last preliminary notion towards the parametric execution semantics of GKABs, we specify when a state $\langle A, m, \delta \rangle$ can be considered to be *final* (i.e., the execution of δ can be considered completed), written $\langle A, m, \delta \rangle \in \mathbb{F}$. This is done by defining the set \mathbb{F} of final states as follows:

Final State **Definition 4.14** (Final State). Let $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ be a GKAB. We define the set \mathbb{F} of *final states of \mathcal{G}* as the least set of states of the form $\langle A, m, \delta \rangle$ such that A is an ABox over T , m is a service call map, δ' is a program over Γ , and the following hold:

1. $\langle A, m, \varepsilon \rangle \in \mathbb{F}$;
2. $\langle A, m, \delta_1 | \delta_2 \rangle \in \mathbb{F}$ if $\langle A, m, \delta_1 \rangle \in \mathbb{F}$ or $\langle A, m, \delta_2 \rangle \in \mathbb{F}$;
3. $\langle A, m, \delta_1 ; \delta_2 \rangle \in \mathbb{F}$ if $\langle A, m, \delta_1 \rangle \in \mathbb{F}$ and $\langle A, m, \delta_2 \rangle \in \mathbb{F}$;
4. $\langle A, m, \text{if } \varphi \text{ then } \delta_1 \text{ else } \delta_2 \rangle \in \mathbb{F}$
if $\text{ASK}(\varphi, T, A) = \text{true}$, and $\langle A, m, \delta_1 \rangle \in \mathbb{F}$;
5. $\langle A, m, \text{if } \varphi \text{ then } \delta_1 \text{ else } \delta_2 \rangle \in \mathbb{F}$
if $\text{ASK}(\varphi, T, A) = \text{false}$, and $\langle A, m, \delta_2 \rangle \in \mathbb{F}$;
6. $\langle A, m, \text{while } \varphi \text{ do } \delta \rangle \in \mathbb{F}$ if $\text{ASK}(\varphi, T, A) = \text{false}$;
7. $\langle A, m, \text{while } \varphi \text{ do } \delta \rangle \in \mathbb{F}$ if $\text{ASK}(\varphi, T, A) = \text{true}$, and $\langle A, m, \delta \rangle \in \mathbb{F}$.

■

Now, given a filter relation f , we define the *program execution relation* $\xrightarrow{\alpha\sigma, f}$, describing how a grounded action simultaneously evolves the data- and program-state.

Definition 4.15 (Program Execution Relation). Given a GKAB $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$, and a filter relation f , we define a *program execution relation* $\xrightarrow{\alpha\sigma, f}$ as follows: Program Execution Relation

1. $\langle A, m, \mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p}) \rangle \xrightarrow{\alpha\sigma, f} \langle A', m', \varepsilon \rangle$,
if $\langle \langle A, m \rangle, \alpha\sigma, \langle A', m' \rangle \rangle \in \text{TELL}_f$, and σ is a legal parameter assignment for α in A w.r.t. $\mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p})$;
2. $\langle A, m, \delta_1 | \delta_2 \rangle \xrightarrow{\alpha\sigma, f} \langle A', m', \delta' \rangle$,
if $\langle A, m, \delta_1 \rangle \xrightarrow{\alpha\sigma, f} \langle A', m', \delta' \rangle$ or $\langle A, m, \delta_2 \rangle \xrightarrow{\alpha\sigma, f} \langle A', m', \delta' \rangle$;
3. $\langle A, m, \delta_1; \delta_2 \rangle \xrightarrow{\alpha\sigma, f} \langle A', m', \delta'_1; \delta_2 \rangle$,
if $\langle A, m, \delta_1 \rangle \xrightarrow{\alpha\sigma, f} \langle A', m', \delta'_1 \rangle$;
4. $\langle A, m, \delta_1; \delta_2 \rangle \xrightarrow{\alpha\sigma, f} \langle A', m', \delta'_2 \rangle$,
if $\langle A, m, \delta_1 \rangle \in \mathbb{F}$, and $\langle A, m, \delta_2 \rangle \xrightarrow{\alpha\sigma, f} \langle A', m', \delta'_2 \rangle$;
5. $\langle A, m, \mathbf{if} \ \varphi \ \mathbf{then} \ \delta_1 \ \mathbf{else} \ \delta_2 \rangle \xrightarrow{\alpha\sigma, f} \langle A', m', \delta'_1 \rangle$,
if $\text{ASK}(\varphi, T, A) = \text{true}$, and $\langle A, m, \delta_1 \rangle \xrightarrow{\alpha\sigma, f} \langle A', m', \delta'_1 \rangle$;
6. $\langle A, m, \mathbf{if} \ \varphi \ \mathbf{then} \ \delta_1 \ \mathbf{else} \ \delta_2 \rangle \xrightarrow{\alpha\sigma, f} \langle A', m', \delta'_2 \rangle$,
if $\text{ASK}(\varphi, T, A) = \text{false}$, and $\langle A, m, \delta_2 \rangle \xrightarrow{\alpha\sigma, f} \langle A', m', \delta'_2 \rangle$;
7. $\langle A, m, \mathbf{while} \ \varphi \ \mathbf{do} \ \delta \rangle \xrightarrow{\alpha\sigma, f} \langle A', m', \delta'; \mathbf{while} \ \varphi \ \mathbf{do} \ \delta \rangle$,
if $\text{ASK}(\varphi, T, A) = \text{true}$, and $\langle A, m, \delta \rangle \xrightarrow{\alpha\sigma, f} \langle A', m', \delta' \rangle$.

■

We are now defining the construction of GKABs transition systems that is parameterized with a filter as follows.

Definition 4.16 (GKAB Transition System). Given a GKAB $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ and a filter relation f , we define the *transition system of \mathcal{G} w.r.t. f* , written $\Upsilon_{\mathcal{G}}^f$, as GKAB Transition System

$\langle \Delta, T, \Sigma, s_0, \text{abox}, \Rightarrow \rangle$, where

1. $s_0 = \langle A_0, \emptyset, \delta \rangle$, and
2. Σ and \Rightarrow are defined by simultaneous induction as the smallest sets such that
 - a) $s_0 \in \Sigma$, and
 - b) if $\langle A, m, \delta \rangle \in \Sigma$ and $\langle A, m, \delta \rangle \xrightarrow{\alpha\sigma, f} \langle A', m', \delta' \rangle$, then $\langle A', m', \delta' \rangle \in \Sigma$ and $\langle A, m, \delta \rangle \Rightarrow \langle A', m', \delta' \rangle$.

■

By suitably concretizing the filter relation, we can obtain various execution semantics for GKABs. We are now exploiting filter relations to define the standard execution semantics of GKAB. In particular, we define a filter relation f_S as follows:

Definition 4.17 (Standard Filter f_S). Let $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ be a GKAB. Given an ABox A , an atomic action $\alpha \in \Gamma$, a legal parameter assignment σ for α in A w.r.t. a certain action invocation $\mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p})$ in δ , and a service call substitution $\theta \in \text{EVAL}(\text{ADD}(T, A, \alpha\sigma))$, let $F^+ = \text{ADD}(T, A, \alpha\sigma)\theta$ and $F^- = \text{DEL}(T, A, \alpha\sigma)$. We then have $\langle A, F^+, F^-, A' \rangle \in f_S$ if $A' = (A \setminus F^-) \cup F^+$, Standard Filter f_S

■

Filter f_S gives rise to the *standard execution semantics* for \mathcal{G} . Essentially it just applies the update induced by the ground atomic action $\alpha\sigma$ (giving priority to additions over deletions). We call the GKABs adopting these semantics *Standard GKABs* (*S-GKABs*).

Definition 4.18 (GKAB Standard Transition System). Given a GKAB $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ and a standard filter f_S , the *standard transition system of \mathcal{G}* , written $\Upsilon_{\mathcal{G}}^{f_S}$, is the transition system of \mathcal{G} w.r.t. f_S . ■

An important observation in S-GKABs is that they reject those actions that lead into an inconsistent state. This is the case because a tuple $\langle A, F^+, F^-, A' \rangle \in f_S$ might have the ABox A' T -inconsistent. However, each tuple $\langle \langle A, m \rangle, \alpha\sigma, \langle A', m' \rangle \rangle \in \text{TELL}_{f_S}$ requires that A and A' are T -consistent.

The notion of run and run-boundedness of S-GKABs transition systems is defined similarly as in Definitions 3.27 and 3.28.

Example 4.19. Consider our specification of GKAB $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ in Example 4.3. Let \mathcal{G} be an S-GKABs (i.e., it adopts standard execution semantics and its execution semantics is provided by the standard transition system as in Definition 4.18). In the following, we give the intuition of how a program is executed, how the system is progressing, and how the standard transition system is constructed.

The construction of the $\Upsilon_{\mathcal{G}}^{f_S}$ is started from the initial state $s_0 = \langle A_0, m_0, \delta \rangle$, where

$$\delta = \text{while } \exists x. [\text{Order}(x)] \wedge \neg [\text{DeliveredOrder}(x)] \text{ do } \delta_0$$

and $m_0 = \emptyset$. Since there exists an *order* in A_0 (note that $A_0 = \{\text{ReceivedOrder}(\text{chair}), \text{ApprovedOrder}(\text{table})\}$), we enter the loop and execute δ_0 that is a sequence of program $\delta_1; \delta_2; \delta_3; \delta_4; \delta_5$. We then need to execute δ_1 that is basically an if-else conditional statement as follows:

$$\begin{aligned} \delta_1 = & \text{if } \neg [\exists x. \text{ApprovedOrder}(x)] \\ & \text{then pick } [\text{ReceivedOrder}(x)].\text{approveOrder}(x) \\ & \text{else } \varepsilon \end{aligned}$$

Since there exists an *approved order* in A_0 (note that $\text{ApprovedOrder}(\text{table}) \in A_0$), and the else part of δ_1 is ε (i.e., $\langle A_0, m_0, \varepsilon \rangle \in \mathbb{F}$), we then have $\langle A_0, m_0, \delta_1 \rangle \in \mathbb{F}$. Then we need to execute δ_2 that is an action invocation **pick true.prepareOrders()**. Notice that **prepareOrders** is executable in A_0 with a legal parameter assignment σ where σ is an empty substitution. Since we consider standard filter relation, we basically have the following:

- $\text{ADD}(T, A_0, \text{prepareOrders}\sigma) = \{$
 $\quad \text{designedBy}(\text{table}, \text{GETDESIGNER}(\text{table})),$
 $\quad \text{Designer}(\text{GETDESIGNER}(\text{table})),$
 $\quad \text{hasDesign}(\text{table}, \text{GETDESIGN}(\text{table})),$
 $\quad \text{hasAssemblingLoc}(\text{table}, \text{ASSIGNASSEMBLINGLOC}(\text{table}))$
 $\quad \}$
- $\text{DEL}(T, A_0, \text{prepareOrders}\sigma) = \emptyset$.
- $\text{EVAL}(\text{ADD}(T, A_0, \text{prepareOrders}\sigma))$ contains infinite set of substitutions where each substitution substitutes the service calls $\text{GETDESIGNER}(\text{table})$, $\text{GETDESIGN}(\text{table})$, and $\text{ASSIGNASSEMBLINGLOC}(\text{table})$ into constants in Δ .

and we have infinite tuple of the form

$$\langle A_0, \text{ADD}(T, A_0, \text{prepareOrders}\sigma)\theta, \text{DEL}(T, A_0, \text{prepareOrders}\sigma), A_1 \rangle$$

in f_S , where

$$A_1 = (A_0 \setminus \text{DEL}(T, A_0, \text{prepareOrders}\sigma)) \cup \text{ADD}(T, A_0, \text{prepareOrders}\sigma)\theta$$

and $\theta \in \text{EVAL}(\text{ADD}(T, A_0, \text{prepareOrders}\sigma))$. I.e., we basically have that A_1 is of the form

$$A_0 \cup \left\{ \begin{array}{l} \text{designedBy}(\text{table}, \text{GETDESIGNER}(\text{table})), \\ \text{Designer}(\text{GETDESIGNER}(\text{table})), \\ \text{hasDesign}(\text{table}, \text{GETDESIGN}(\text{table})), \\ \text{hasAssemblingLoc}(\text{table}, \text{ASSIGNASSEMBLINGLOC}(\text{table})) \end{array} \right\}$$

where $\text{GETDESIGNER}(\text{table})$, $\text{GETDESIGN}(\text{table})$, and $\text{ASSIGNASSEMBLINGLOC}(\text{table})$ are arbitrarily substituted with constants from Δ . Furthermore, we have infinite tuple of the form

$$\langle \langle A_0, m_0 \rangle, \alpha\sigma, \langle A_1, m_1 \rangle \rangle$$

in TELL_{f_S} where A_1 is of the form as above, and m_1 substitutes the service calls $\text{GETDESIGNER}(\text{table})$, $\text{GETDESIGN}(\text{table})$, and $\text{ASSIGNASSEMBLINGLOC}(\text{table})$ with the corresponding constants. Therefore, based on the definition on how the program are executed (see Definition 4.15), basically we have infinitely many successors for s_0 , each of the form $\langle A_1, m_1, \delta' \rangle$ where

1. $\delta' = \delta'_0$; **while** $\exists x. [\text{Order}(x)] \wedge \neg [\text{DeliveredOrder}(x)]$ **do** δ_0 ,
2. $\delta'_0 = \delta_3; \delta_4; \delta_5$,
3. A_1 and m_1 are as above.

Next, the execution of S-GKAB is continued by applying the same procedure to all successors of s_0 and so on.

Rejecting Inconsistent States.

We now provide an example where S-GKABs reject inconsistent states. Continuing our example, consider a particular sucessor state of s_0 , namely state $s_1 = \langle A_1, m_1, \delta' \rangle$, where

- $A_1 = \{ \text{ReceivedOrder}(\text{chair}), \text{ApprovedOrder}(\text{table}), \text{designedBy}(\text{table}, \text{alice}), \text{Designer}(\text{alice}), \text{hasDesign}(\text{table}, \text{ecodesign}), \text{hasAssemblingLoc}(\text{table}, \text{bolzano}) \}$,
- $m_1 = \{ [\text{GETDESIGNER}(\text{table}) \rightarrow \text{alice}], [\text{GETDESIGN}(\text{table}) \rightarrow \text{ecodesign}], [\text{ASSIGNASSEMBLINGLOC}(\text{table}) \rightarrow \text{bolzano}] \}$,
- $\delta' = \delta_3; \delta_4; \delta_5$; **while** $\exists x. [\text{Order}(x)] \wedge \neg [\text{DeliveredOrder}(x)]$ **do** δ_0 .

The next step is to execute δ_3 that is an action invocation of the form **pick** `true.assembleOrders()`. The execution of action `assembleOrders` involves the service calls `GETASSEMBLER/1` and `GETASSEMBLINGLOC/1`. Thus it is easy to see that there are infinite successor states of s_1 each of the form $\langle A_2, m_2, \delta'' \rangle$, where A_2 is of the form as follows:

$$A_2 = (A_1 \setminus \{\text{ApprovedOrder}(\text{table})\}) \cup \{\text{AssembledOrder}(\text{table}), \\ \text{assembledBy}(\text{table}, \text{GETASSEMBLER}(\text{table})), \\ \text{Assembler}(\text{GETASSEMBLER}(\text{table})), \\ \text{hasAssemblingLoc}(\text{table}, \text{GETASSEMBLINGLOC}(\text{table})) \}$$

in which `GETASSEMBLINGLOC(table)` as well as `GETASSEMBLER(table)` are arbitrarily substituted with constants from Δ by a substitution $\theta \in \text{EVAL}(\text{ADD}(T, A_1, \text{assembleOrders}\sigma))$ and $m_2 = m_1 \cup \theta$. Moreover, we have

$$\delta'' = \delta_4; \delta_5; \textbf{while } \exists x. [\text{Order}(x)] \wedge \neg [\text{DeliveredOrder}(x)] \textbf{ do } \delta_0.$$

Now, as an example of a successor that causes inconsistency, consider a possible substitution of `GETASSEMBLINGLOC(table)` into “trento” and `GETASSEMBLER(table)` into “alice” by a particular substitution $\theta \in \text{EVAL}(\text{ADD}(T, A_1, \text{assembleOrders}\sigma))$. We then have a state $s_2 = \langle A_2, m_2, \delta'' \rangle$ where

- $A_2 = \{ \text{ReceivedOrder}(\text{chair}), \text{designedBy}(\text{table}, \text{alice}), \text{Designer}(\text{alice}), \\ \text{hasDesign}(\text{table}, \text{ecodesign}), \text{hasAssemblingLoc}(\text{table}, \text{bolzano}), \\ \text{AssembledOrder}(\text{table}), \text{assembledBy}(\text{table}, \text{alice}), \\ \text{Assembler}(\text{alice}), \text{hasAssemblingLoc}(\text{table}, \text{trento}) \},$
- $m_2 = \{ [\text{GETDESIGNER}(\text{table}) \rightarrow \text{alice}], [\text{GETDESIGN}(\text{table}) \rightarrow \text{ecodesign}], \\ [\text{ASSIGNASSEMBLINGLOC}(\text{table}) \rightarrow \text{bolzano}], \\ [\text{GETASSEMBLER}(\text{table}) \rightarrow \text{alice}], \\ [\text{GETASSEMBLINGLOC}(\text{table}) \rightarrow \text{trento}] \},$
- $\delta'' = \delta_4; \delta_5; \textbf{while } \exists x. [\text{Order}(x)] \wedge \neg [\text{DeliveredOrder}(x)] \textbf{ do } \delta_0.$

Notice that the presence of assertions `Assembler(alice)` and `Designer(alice)` in A_2 trigger the violation of TBox assertion `Designer \sqsubseteq \neg Assembler`, because `Designer` and `Assembler` are two disjoint concepts (i.e., a constant can not belong to both of those concepts at the same time). Moreover, the existence of assertions `hasAssemblingLoc(table, bolzano)` and `hasAssemblingLoc(table, trento)` yields the violation of TBox assertion `(funct hasAssemblingLoc)` because they make the role `hasAssemblingLoc` not functional (i.e., table has two different ranges namely bolzano and trento). Therefore, since in this case A_2 is T -inconsistent, then we have that s_2 is rejected.

4.3 Capturing KABs within Standard GKABs

Here we show that our S-GKABs are able to capture KABs, and show that the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ properties over KABs can be reduced to the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ properties over S-GKABs. The core idea is to invent a generic translation that transforms any KABs into S-GKABs such that their transition systems are “equal” (in the sense that they have the same structure and each corresponding state contains the same ABox and service call map), and hence they should satisfy the same $\mu\mathcal{L}_A^{\text{EQL}}$ formulas.

We now define the translation from KABs to S-GKABs as follows:

Definition 4.20 (Translation from a KAB to an S-GKAB). We define a translation τ_S that, given an KAB $\mathcal{K} = \langle T, A_0, \Gamma, \Pi \rangle$, generates an S-GKAB $\tau_S(\mathcal{K}) = \langle T, A_0, \Gamma, \delta \rangle$ in which program δ is obtained from Π as

Translation from a KAB to an S-GKAB

$$\delta = \mathbf{while\ true\ do\ } (a_1 | a_2 | \dots | a_{|\Pi|}),$$

where, for each condition-action rule $Q_i(\vec{x}) \mapsto \alpha_i(\vec{x}) \in \Pi$, we have $a_i = \mathbf{pick\ } Q_i(\vec{x}).\alpha_i(\vec{x})$. ■

Intuitively, the translation produces a program that continues forever to non-deterministically pick an executable action with parameters (as specified by Π), or stops if no action is executable.

Next, we continue our journey to recast the verification of KABs into S-GKABs by:

1. formalizing the notion of “equality” between transition systems by introducing the notion of E-Bisimulation, as well as showing that two E-bisimilar transition systems can not be distinguished by $\mu\mathcal{L}_A^{\text{EQL}}$ properties (in Section 4.3.1), and
2. showing that τ_S transforms KABs into S-GKABs such that their transition systems are E-bisimilar (in Section 4.3.2).

4.3.1 E-Bisimulation

We now define the notion of *E-Bisimulation* and show that two E-bisimilar transition systems can not be distinguished by a $\mu\mathcal{L}_A^{\text{EQL}}$ formula.

Definition 4.21 (E-Bisimulation).

Let $\mathcal{T}_1 = \langle \Delta, T, \Sigma_1, s_{01}, \text{abox}_1, \Rightarrow_1 \rangle$ and $\mathcal{T}_2 = \langle \Delta, T, \Sigma_2, s_{02}, \text{abox}_2, \Rightarrow_2 \rangle$ be transition systems, with $\text{ADOM}(\text{abox}_1(s_{01})) \subseteq \Delta$ and $\text{ADOM}(\text{abox}_2(s_{02})) \subseteq \Delta$. An *E-Bisimulation* between \mathcal{T}_1 and \mathcal{T}_2 is a relation $\mathcal{B} \subseteq \Sigma_1 \times \Sigma_2$ such that $\langle s_1, s_2 \rangle \in \mathcal{B}$ implies that:

E-Bisimulation

1. $\text{abox}_1(s_1) = \text{abox}_2(s_2)$
2. for each s'_1 , if $s_1 \Rightarrow_1 s'_1$ then there exist s'_2 with $s_2 \Rightarrow_2 s'_2$ such that $\langle s'_1, s'_2 \rangle \in \mathcal{B}$.
3. for each s'_2 , if $s_2 \Rightarrow_2 s'_2$ then there exists s'_1 with $s_1 \Rightarrow_1 s'_1$, such that $\langle s'_1, s'_2 \rangle \in \mathcal{B}$.

■

Let $\mathcal{T}_1 = \langle \Delta, T, \Sigma_1, s_{01}, \text{abox}_1, \Rightarrow_1 \rangle$ and $\mathcal{T}_2 = \langle \Delta, T, \Sigma_2, s_{02}, \text{abox}_2, \Rightarrow_2 \rangle$ be transition systems, a state $s_1 \in \Sigma_1$ is *E-bisimilar* to $s_2 \in \Sigma_2$, written $s_1 \sim_E s_2$, if there exists an E-Bisimulation \mathcal{B} between \mathcal{T}_1 and \mathcal{T}_2 such that $\langle s_1, s_2 \rangle \in \mathcal{B}$. The transition system \mathcal{T}_1 is *E-bisimilar* to \mathcal{T}_2 , written $\mathcal{T}_1 \sim_E \mathcal{T}_2$, if there exists an E-Bisimulation \mathcal{B} between \mathcal{T}_1 and \mathcal{T}_2 such that $\langle s_{01}, s_{02} \rangle \in \mathcal{B}$.

Lemma 4.22. Consider two transition systems $\mathcal{T}_1 = \langle \Delta, T, \Sigma_1, s_{01}, \text{abox}_1, \Rightarrow_1 \rangle$ and $\mathcal{T}_2 = \langle \Delta, T, \Sigma_2, s_{02}, \text{abox}_2, \Rightarrow_2 \rangle$ such that $\mathcal{T}_1 \sim_E \mathcal{T}_2$. For every $\mu\mathcal{L}_A^{\text{EQL}}$ closed formula Φ , we have:

$$\mathcal{T}_1 \models \Phi \text{ if and only if } \mathcal{T}_2 \models \Phi.$$

Proof. The claim easily follows since two E-bisimilar transition systems are essentially equal in terms of the structure and the ABoxes that are contained in each two bisimilar states. \square

4.3.2 Reducing the Verification of KABs to Standard GKABs

Here we show that the transition system of a KAB and the transition system of its corresponding S-GKAB are E-bisimilar. Then, by using the result from the previous subsection we can easily recast the verification problem and hence achieve our purpose.

Lemma 4.23. Let \mathcal{K} be a KAB with transition system $\mathcal{T}_{\mathcal{K}}^S$, and let $\tau_S(\mathcal{K})$ be an S-GKAB with transition system $\mathcal{T}_{\tau_S(\mathcal{K})}^{fs}$ obtained through τ_S . Consider a state $\langle A_k, m_k \rangle$ of $\mathcal{T}_{\mathcal{K}}^S$ and a state $\langle A_g, m_g, \delta_g \rangle$ of $\mathcal{T}_{\tau_S(\mathcal{K})}^{fs}$. If $A_k = A_g$, and $m_k = m_g$, then $\langle A_k, m_k \rangle \sim_E \langle A_g, m_g, \delta_g \rangle$.

Proof. Let

1. $\mathcal{K} = \langle T, A_0, \Gamma, \Pi \rangle$, and
 $\mathcal{T}_{\mathcal{K}}^S = \langle \Delta, T, \Sigma_k, s_{0k}, \text{abox}_k, \Rightarrow_k \rangle$,
2. $\tau_S(\mathcal{K}) = \langle T, A_0, \Gamma, \delta \rangle$, and
 $\mathcal{T}_{\tau_S(\mathcal{K})}^{fs} = \langle \Delta, T, \Sigma_g, s_{0g}, \text{abox}_g, \Rightarrow_g \rangle$.

To prove the lemma, we show that, for every state $\langle A'_k, m'_k \rangle$ s.t. $\langle A_k, m_k \rangle \Rightarrow_k \langle A'_k, m'_k \rangle$, there exists a state $\langle A'_g, m'_g, \delta'_g \rangle$ s.t.:

1. $\langle A_g, m_g, \delta_g \rangle \Rightarrow_g \langle A'_g, m'_g, \delta'_g \rangle$;
2. $A'_k = A'_g$;
3. $m'_k = m'_g$.

By definition of $\mathcal{T}_{\mathcal{K}}^S$, if $\langle A_k, m_k \rangle \Rightarrow \langle A'_k, m'_k \rangle$, then there exist

1. a condition action rule $Q(\vec{p}) \mapsto \alpha(\vec{p})$,
2. an action $\alpha \in \Gamma$ with parameters \vec{p} ,
3. a parameter substitution σ , and
4. a substitution θ .

such that (i) $\theta \in \text{EVAL}(T, A_k, \alpha\sigma)$ and agrees with m_k , (ii) α is executable in state A_k with a parameter substitution σ , (iii) $A'_k = \text{DO}(T, A_k, \alpha\sigma)\theta$, and (iv) $m'_k = m_k \cup \theta$.

Now, since $\delta = \mathbf{while\ true\ do\ } (a_1|a_2|\dots|a_{|\Pi|})$, and each a_i is an action invocation obtained from a condition-action rule in Π , then there exists an action invocation a_i such that $a_i = \mathbf{pick\ } Q(\vec{x}).\alpha(\vec{x})$. Since $A_k = A_g$, and $m_k = m_g$, by considering how a transition is created in the transition system of S-GKABs, it is easy to see that there exists a state $\langle A'_g, m'_g, \delta'_g \rangle$ such that $\langle A_g, m_g, \delta_g \rangle \Rightarrow_g \langle A'_g, m'_g, \delta'_g \rangle$, $A'_g = A'_k$, and $m'_g = m'_k$. Thus, the claim is proven. \square

Lemma 4.24. Given a KAB \mathcal{K} , we have $\mathcal{T}_{\mathcal{K}}^S \sim_E \mathcal{T}_{\tau_S(\mathcal{K})}^{fs}$

Proof. Let

1. $\mathcal{K} = \langle T, A_0, \Gamma, \Pi \rangle$, and
 $\mathcal{Y}_{\mathcal{K}}^S = \langle \Delta, T, \Sigma_k, s_{0k}, \text{abox}_k, \Rightarrow_k \rangle$,
2. $\tau_S(\mathcal{K}) = \langle T, A_0, \Gamma, \delta \rangle$, and
 $\mathcal{Y}_{\tau_S(\mathcal{K})}^{fs} = \langle \Delta, T, \Sigma_g, s_{0g}, \text{abox}_g, \Rightarrow_g \rangle$.

We have that $s_{0k} = \langle A_0, m_k \rangle$ and $s_{0g} = \langle A_0, m_g, \delta \rangle$ where $m_k = m_g = \emptyset$. Hence, by Lemma 4.23, we have $s_{0k} \sim_E s_{0g}$. Therefore, by the definition of E-bisimulation between two transition systems, we have $\mathcal{Y}_{\mathcal{K}}^S \sim_E \mathcal{Y}_{\tau_S(\mathcal{K})}^{fs}$. \square

Having Lemma 4.24 in hand, we can easily show that the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ over KABs can be reduced to the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ over S-GKABs by also making use the result from the previous subsection.

Theorem 4.25. *Verification of $\mu\mathcal{L}_A^{\text{EQL}}$ properties over KABs can be recast as verification over S-GKABs.*

Proof. The proof can be easily obtained since we can translate KABs into S-GKABs using τ_S and then we can easily show that given a KAB \mathcal{K} and a closed $\mu\mathcal{L}_A^{\text{EQL}}$ formula Φ , we have $\mathcal{Y}_{\mathcal{K}}^S \models \Phi$ iff $\mathcal{Y}_{\tau_S(\mathcal{K})}^{fs} \models \Phi$ due to the fact that by Lemma 4.24, we have that $\mathcal{Y}_{\mathcal{K}}^S \sim_E \mathcal{Y}_{\tau_S(\mathcal{K})}^{fs}$ and hence the claim is directly follows from Lemma 4.22. \square

4.4 Verification of Standard GKABs (S-GKABs)

The problem definition of the $\mu\mathcal{L}_A^{\text{EQL}}$ formula verification over S-GKABs is defined similarly as in KABs (see Definition 3.13).

Example 4.26. Continuing our running example, an example of $\mu\mathcal{L}_A^{\text{EQL}}$ properties to be verified is as follows:

$$\nu Z. (\forall x. \text{Order}(x) \rightarrow \mu Y. (\text{DeliveredOrder}(x) \vee \langle \neg \rangle Y)) \wedge [-] Z$$

Intuitively, this formula says that, along every path, it is always true that each order x will be eventually delivered.

Here, we solve the problem of S-GKABs verification by compiling S-GKABs into KABs and show that the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ formulas over S-GKAB can be recast as verification over KAB. (This claim is formally stated in Theorem 4.54). To this aim, technically we do the following:

1. We define a special bisimulation relation between two transition system namely *jumping bisimulation* (see Section 4.4.1).
2. Furthermore, also in Section 4.4.1, we define a generic translation t_j that takes a $\mu\mathcal{L}_A^{\text{EQL}}$ formula Φ in Negative Normal Form (NNF) as an input and produces a $\mu\mathcal{L}_A^{\text{EQL}}$ formula $t_j(\Phi)$, and then we also show that two jumping bisimilar transition system can not be distinguished by any $\mu\mathcal{L}_A^{\text{EQL}}$ formula (in NNF) modulo the translation t_j .
3. In Section 4.4.2, we define a generic translation $\tau_{\mathcal{G}}$, that given an S-GKAB \mathcal{G} , produces a KAB $\tau_{\mathcal{G}}(\mathcal{G})$. The core idea of this translation is to transform the

given program δ and the set of actions in S-GKAB \mathcal{G} into a process (a set of condition-action rules) and a set of KAB actions, such that all possible sequence of action executions that is enforced by δ can be mimicked by the process in KAB (which determines all possible sequence of action executions in KAB).

4. In the Section 4.4.3, we show that the transition system of a GKAB \mathcal{G} and the transition system of its corresponding KAB $\tau_{\mathcal{G}}(\mathcal{G})$ (obtained through translation $\tau_{\mathcal{G}}$) are bisimilar w.r.t. the jumping bisimulation relation.
5. Making use all of the ingredients above, in the end we show that a GKAB \mathcal{G} satisfies a certain $\mu\mathcal{L}_A^{\text{EQL}}$ formula Φ if and only if its corresponding KAB $\tau_{\mathcal{G}}(\mathcal{G})$ satisfies a $\mu\mathcal{L}_A^{\text{EQL}}$ formula $t_j(\Phi)$ (see Section 4.4.3).

Special Markers

For a technical reason, we reserve some fresh concept names **Flag**, **Noop** and **State** (i.e., they are outside of any TBox vocabulary), and they are not allowed to be used in any temporal properties (i.e., in $\mu\mathcal{L}_A^{\text{EQL}}$ or $\mu\mathcal{L}_A$ formulas). We call them *special marker concept names*. Additionally, we make use the constants in Δ_0 to populate them. We call *special marker* an ABox assertion that is obtained by applying either **Flag**, **Noop** or **State** to a constant in Δ_0 . Additionally, we call *flag* a special marker formed by applying either concept name **Flag** or **Noop** to a constant in Δ_0 . Later on, we use flags as markers to impose a certain sequence of action executions, and we use a special marker **State**(*temp*) (where *temp* $\in \Delta_0$) to mark an *intermediate state*.

4.4.1 Jumping Bisimulation (J-Bisimulation)

As a start towards defining the notion of J-Bisimulation, we introduce the notion of equality modulo flag between two ABoxes as follows:

Equal Modulo Special Markers

Definition 4.27 (Equal Modulo Special Markers). Given a TBox T , two ABoxes A_1 and A_2 over $\text{voc}(T)$ that might contain special markers, we say A_1 *equal to* A_2 *modulo special markers*, written $A_1 \simeq A_2$ (or equivalently $A_2 \simeq A_1$), if the following hold:

- For each concept name $N \in \text{voc}(T)$ (i.e., N is not a special marker concept name), we have a concept assertion $N(c) \in A_1$ if and only if a concept assertion $N(c) \in A_2$,
- For each role name $P \in \text{voc}(T)$, we have a role assertion $P(c_1, c_2) \in A_1$ if and only if a role assertion $P(c_1, c_2) \in A_2$.

■

Essentially, the definition Definition 4.27 above says that two ABoxes are equal modulo special markers if they contain the same ABox assertions except the special markers. Next, we show some interesting properties related to the notion of equality modulo flag between two ABoxes.

Lemma 4.28. $A_1 = A_2$ *implies* $A_1 \simeq A_2$.

Proof. Trivially true from the definition of $A_1 \simeq A_2$ above (see Definition 4.27) since both A_1 and A_2 contain the same set of ABox assertions. \square

Lemma 4.29. *Given a GKAB $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$, two ABoxes A_1 and A_2 over $\text{voc}(T)$ which might contain special markers, and an ECQ Q over $\langle T, A_0 \rangle$ which does not*

contain any atoms whose predicates are special marker concept names. We have that if $A_1 \simeq A_2$, then $\text{CERT}(Q, T, A_1) = \text{CERT}(Q, T, A_2)$.

Proof. Trivially hold since without considering special markers, we have that a concept assertion $N(c) \in A_1$ if and only if a concept assertion $N(c) \in A_2$, and also a role assertion $P(c_1, c_2) \in A_1$ if and only if a role assertion $P(c_1, c_2) \in A_2$ (i.e., we can consider that $A_1 = A_2$ because we do not query the special marker). Hence $\text{CERT}(Q, T, A_1) = \text{CERT}(Q, T, A_2)$. \square

We now proceed to define the notion of *jumping bisimulation* as follows.

Definition 4.30 (Jumping Bisimulation (J-Bisimulation)).

*Jumping Bisimulation
(J-Bisimulation)*

Let $\mathcal{T}_1 = \langle \Delta, T, \Sigma_1, s_{01}, \text{abox}_1, \Rightarrow_1 \rangle$ and $\mathcal{T}_2 = \langle \Delta, T, \Sigma_2, s_{02}, \text{abox}_2, \Rightarrow_2 \rangle$ be KB transition systems, with $\text{ADOM}(\text{abox}_1(s_{01})) \subseteq \Delta$ and $\text{ADOM}(\text{abox}_2(s_{02})) \subseteq \Delta$. A *jumping bisimulation* (J-Bisimulation) between \mathcal{T}_1 and \mathcal{T}_2 is a relation $\mathcal{B} \subseteq \Sigma_1 \times \Sigma_2$ such that $\langle s_1, s_2 \rangle \in \mathcal{B}$ implies that:

1. $\text{abox}_1(s_1) \simeq \text{abox}_2(s_2)$
2. for each s'_1 , if $s_1 \Rightarrow_1 s'_1$ then there exist s'_2, t_1, \dots, t_n (for $n \geq 0$) with

$$s_2 \Rightarrow_2 t_1 \Rightarrow_2 \dots \Rightarrow_2 t_n \Rightarrow_2 s'_2$$

such that $\langle s'_1, s'_2 \rangle \in \mathcal{B}$, $\text{State}(\text{temp}) \notin \text{abox}_2(s'_2)$ and $\text{State}(\text{temp}) \in \text{abox}_2(t_i)$ for $i \in \{1, \dots, n\}$.

3. for each s'_2 , if

$$s_2 \Rightarrow_2 t_1 \Rightarrow_2 \dots \Rightarrow_2 t_n \Rightarrow_2 s'_2$$

(for $n \geq 0$) with $\text{State}(\text{temp}) \in \text{abox}_2(t_i)$ for $i \in \{1, \dots, n\}$ and $\text{State}(\text{temp}) \notin \text{abox}_2(s'_2)$, then there exists s'_1 with $s_1 \Rightarrow_1 s'_1$, such that $\langle s'_1, s'_2 \rangle \in \mathcal{B}$. \blacksquare

Given two KB transition systems $\mathcal{T}_1 = \langle \Delta, T, \Sigma_1, s_{01}, \text{abox}_1, \Rightarrow_1 \rangle$ and $\mathcal{T}_2 = \langle \Delta, T, \Sigma_2, s_{02}, \text{abox}_2, \Rightarrow_2 \rangle$, a state $s_1 \in \Sigma_1$ is *J-bisimilar* to $s_2 \in \Sigma_2$, written $s_1 \sim_J s_2$, if there exists a jumping bisimulation \mathcal{B} between \mathcal{T}_1 and \mathcal{T}_2 such that $\langle s_1, s_2 \rangle \in \mathcal{B}$. A transition system \mathcal{T}_1 is *J-bisimilar* to \mathcal{T}_2 , written $\mathcal{T}_1 \sim_J \mathcal{T}_2$, if there exists a jumping bisimulation \mathcal{B} between \mathcal{T}_1 and \mathcal{T}_2 such that $\langle s_{01}, s_{02} \rangle \in \mathcal{B}$.

Now, in the Lemma 4.32 below, we show that two transition systems which are J-bisimilar can not be distinguished by any $\mu\mathcal{L}_A^{\text{EQL}}$ formula (in NNF) modulo a translation t_j which is defined as follows:

Definition 4.31 (Translation t_j). We define a *translation* t_j that transforms an arbitrary $\mu\mathcal{L}_A^{\text{EQL}}$ formula Φ (in NNF) into another $\mu\mathcal{L}_A^{\text{EQL}}$ formula Φ' inductively by recurring over the structure of Φ as follows:

- $t_j(Q) = Q$
- $t_j(\neg Q) = \neg Q$
- $t_j(Qx.\Phi) = Qx.t_j(\Phi)$
- $t_j(\Phi_1 \circ \Phi_2) = t_j(\Phi_1) \circ t_j(\Phi_2)$
- $t_j(\odot Z.\Phi) = \odot Z.t_j(\Phi)$
- $t_j(\langle \rightarrow \rangle \Phi) = \langle \rightarrow \rangle \mu Z.((\text{State}(\text{temp}) \wedge \langle \rightarrow \rangle Z) \vee (\neg \text{State}(\text{temp}) \wedge t_j(\Phi)))$
- $t_j([\rightarrow] \Phi) = [\rightarrow] \mu Z.((\text{State}(\text{temp}) \wedge [\rightarrow] Z \wedge \langle \rightarrow \rangle \top) \vee (\neg \text{State}(\text{temp}) \wedge t_j(\Phi)))$

where:

- \circ is a binary operator (\vee , \wedge , \rightarrow , or \leftrightarrow),
- \odot is least (μ) or greatest (ν) fix-point operator,
- \mathcal{Q} is forall (\forall) or existential (\exists) quantifier.

■

In brief, t_j translates a given formula into a formula that skips the states in which $\text{State}(temp)$ hold (i.e., bypass the intermediate states). To better understand the translation t_j , we provide some more intuitions of it as follows:

- the formula

$$\mu Z.((\text{State}(temp) \wedge \langle \rightarrow \rangle Z) \vee (\neg \text{State}(temp) \wedge t_j(\Phi)))$$

in the translation $t_j(\langle \rightarrow \rangle \Phi)$ essentially can be also expressed in CTL as

$$\exists[\text{State}(temp) \text{ U } (\neg \text{State}(temp) \wedge t_j(\Phi))]$$

where “U” is the typical CTL “until” operator, and this formula says that there exists a path in which $\text{State}(temp)$ holds until there is a state in which $(\neg \text{State}(temp) \wedge t_j(\Phi))$ holds (See also the translation from CTL into μ -Calculus in [43]). Combining with $\langle \rightarrow \rangle$, we have that intuitively t_j translates $\langle \rightarrow \rangle \Phi$ into a formula stating that there exists a successor state, such that from that successor state, $\text{State}(temp)$ holds until there is a state in which $(\neg \text{State}(temp) \wedge t_j(\Phi))$ holds. Therefore, intuitively, the translation t_j translates $\langle \rightarrow \rangle \Phi$ into a formula saying that there exists a path leading us into a state where $t_j(\Phi)$ holds, and until we reach that state, might need to pass/skip several intermediate states that are marked by $\text{State}(temp)$.

- The intuition for the translation of $[-] \Phi$ is similar to the translation of $\langle \rightarrow \rangle \Phi$ by also noticing that the formula

$$\mu Z.((\text{State}(temp) \wedge [-]Z \wedge \langle \rightarrow \rangle \top) \vee (\neg \text{State}(temp) \wedge t_j(\Phi)))$$

can be also expressed in CTL as

$$\forall[\text{State}(temp) \text{ U } (\neg \text{State}(temp) \wedge t_j(\Phi))].$$

(See also the translation from CTL into μ -Calculus in [43]).

- Later on, we will see that we use $\text{State}(temp)$ to mark the intermediate states in our transition systems, and those intermediate states capture the intermediate results of some computation for generating the “real” successor states. Thus, we are not supposed to query this state and just skip these states.

Lemma 4.32. *Consider two KB transition systems $\mathcal{T}_1 = \langle \Delta, T, \Sigma_1, s_{01}, abox_1, \Rightarrow_1 \rangle$ and $\mathcal{T}_2 = \langle \Delta, T, \Sigma_2, s_{02}, abox_2, \Rightarrow_2 \rangle$, two states $s_1 \in \Sigma_1$ and $s_2 \in \Sigma_2$ such that $s_1 \sim_j s_2$. Then for every formula Φ of $\mu\mathcal{L}_A^{\text{EQL}}$ (in NNF), and every valuations v_1 and v_2 that assign to each of its free variables a constant $c_1 \in \text{ADOM}(abox_1(s_1))$ and $c_2 \in \text{ADOM}(abox_2(s_2))$, such that $c_1 = c_2$, we have that*

$$\mathcal{T}_1, s_1 \models \Phi v_1 \text{ if and only if } \mathcal{T}_2, s_2 \models t_j(\Phi) v_2.$$

Proof. The proof is then organized in three parts:

- (1) We prove the claim for formulae of $\mathcal{L}_A^{\text{EQL}}$, obtained from $\mu\mathcal{L}_A^{\text{EQL}}$ by dropping the predicate variables and the fixpoint constructs. $\mathcal{L}_A^{\text{EQL}}$ corresponds to a first-order variant of the Hennessy Milner logic.
- (2) We extend the results to the infinitary logic obtained by extending $\mathcal{L}_A^{\text{EQL}}$ with arbitrary countable disjunction.
- (3) We recall that fixpoints can be translated into this infinitary logic (cf. [170]), thus proving that the theorem holds for $\mu\mathcal{L}_A^{\text{EQL}}$.

Proof for $\mathcal{L}_A^{\text{EQL}}$. We proceed by induction on the structure of Φ , without considering the case of predicate variable and of fixpoint constructs, which are not part of $\mathcal{L}_A^{\text{EQL}}$.

Base case:

- ($\Phi = Q$). Since $s_1 \sim_J s_2$, we have $\text{abox}_1(s_1) \simeq \text{abox}_2(s_2)$. Hence, since we also restrict that any $\mu\mathcal{L}_A^{\text{EQL}}$ formulas does not use special marker concept names, by Lemma 4.29, we have $\text{CERT}(Q, T, \text{abox}_1(s_1)) = \text{CERT}(Q, T, \text{abox}_2(s_2))$. Hence, since $t_j(Q) = Q$, for every valuations v_1 and v_2 that assign to each of its free variables a constant $c_1 \in \text{ADOM}(\text{abox}_1(s_1))$ and $c_2 \in \text{ADOM}(\text{abox}_2(s_2))$, such that $c_1 = c_2$, we have

$$\mathcal{I}_1, s_1 \models Qv_1 \text{ if and only if } \mathcal{I}_2, s_2 \models t_j(Q)v_2.$$

- ($\Phi = \neg Q$). Similar to the previous case.

Inductive step:

- ($\Phi = \Psi_1 \wedge \Psi_2$). $\mathcal{I}_1, s_1 \models (\Psi_1 \wedge \Psi_2)v_1$ if and only if either $\mathcal{I}_1, s_1 \models \Psi_1v_1$ or $\mathcal{I}_1, s_1 \models \Psi_2v_1$. By induction hypothesis, we have for every valuations v_1 and v_2 that assign to each of its free variables a constant $c_1 \in \text{ADOM}(\text{abox}_1(s_1))$ and $c_2 \in \text{ADOM}(\text{abox}_2(s_2))$, such that $c_1 = c_2$, we have

- $\mathcal{I}_1, s_1 \models \Psi_1v_1$ if and only if $\mathcal{I}_2, s_2 \models t_j(\Psi_1)v_2$, and also
- $\mathcal{I}_1, s_1 \models \Psi_2v_1$ if and only if $\mathcal{I}_2, s_2 \models t_j(\Psi_2)v_2$.

Hence, $\mathcal{I}_1, s_1 \models \Psi_1v_1$ and $\mathcal{I}_1, s_1 \models \Psi_2v_1$ if and only if $\mathcal{I}_2, s_2 \models t_j(\Psi_1)v_2$ and $\mathcal{I}_2, s_2 \models t_j(\Psi_2)v_2$. Therefore we have $\mathcal{I}_1, s_1 \models (\Psi_1 \wedge \Psi_2)v_1$ if and only if $\mathcal{I}_2, s_2 \models (t_j(\Psi_1) \wedge t_j(\Psi_2))v_2$. Since $t_j(\Psi_1 \wedge \Psi_2) = t_j(\Psi_1) \wedge t_j(\Psi_2)$, we have

$$\mathcal{I}_1, s_1 \models (\Psi_1 \wedge \Psi_2)v_1 \text{ if and only if } \mathcal{I}_2, s_2 \models t_j(\Psi_1 \wedge \Psi_2)v_2$$

The proof for the case of $\Phi = \Psi_1 \vee \Psi_2$, $\Phi = \Psi_1 \rightarrow \Psi_2$, and $\Phi = \Psi_1 \leftrightarrow \Psi_2$ can be done similarly.

- ($\Phi = \langle \rightarrow \rangle \Psi$). Assume $\mathcal{I}_1, s_1 \models (\langle \rightarrow \rangle \Psi)v_1$, where v_1 is a valuation that assigns to each free variable of Ψ a constant $c_1 \in \text{ADOM}(\text{abox}_1(s_1))$. Then there exists s'_1 s.t. $s_1 \Rightarrow_1 s'_1$ and $\mathcal{I}_1, s'_1 \models \Psi v_1$. Since $s_1 \sim_J s_2$, there exists s'_2, t_1, \dots, t_n (for $n \geq 0$) with

$$s_2 \Rightarrow_2 t_1 \Rightarrow_2 \dots \Rightarrow_2 t_n \Rightarrow_2 s'_2$$

such that $s'_1 \sim_J s'_2$, $\text{State}(\text{temp}) \in \text{abox}_2(t_i)$ for $i \in \{1, \dots, n\}$, and $\text{State}(\text{temp}) \notin \text{abox}_2(s'_2)$. Hence, by induction hypothesis, for every valuations v_2 that assign to each free variables x of $t_j(\Psi)$ a constant $c_2 \in \text{ADOM}(\text{abox}_2(s_2))$, such that $c_1 = c_2$ and $x/c_1 \in v_1$, we have $\mathcal{I}_2, s'_2 \models t_j(\Psi)v_2$. Consider that

$$s_2 \Rightarrow_2 t_1 \Rightarrow_2 \dots \Rightarrow_2 t_n \Rightarrow_2 s'_2$$

(for $n \geq 0$), $\text{State}(\text{temp}) \in \text{abox}_2(t_i)$ for $i \in \{1, \dots, n\}$, and $\text{State}(\text{temp}) \notin \text{abox}_2(s'_2)$. We then obtain

$$\mathcal{I}_2, s_2 \models (\langle \neg \rangle \mu Z. ((\text{State}(\text{temp}) \wedge \langle \neg \rangle Z) \vee (\neg \text{State}(\text{temp}) \wedge t_j(\Psi)))) v_2.$$

Since $t_j(\langle \neg \rangle \Phi) = \langle \neg \rangle \mu Z. ((\text{State}(\text{temp}) \wedge \langle \neg \rangle Z) \vee (\neg \text{State}(\text{temp}) \wedge t_j(\Phi)))$, we have

$$\mathcal{I}_2, s_2 \models t_j(\langle \neg \rangle \Psi) v_2.$$

For the other direction, assume $\mathcal{I}_2, s_2 \models t_j(\langle \neg \rangle \Psi) v_2$, where v_2 is a valuation that assigns to each free variable of $t_j(\langle \neg \rangle \Psi)$ a constant $c_2 \in \text{ADOM}(\text{abox}_2(s_2))$. By the definition of t_j , we have

$$\mathcal{I}_2, s_2 \models \langle \neg \rangle \mu Z. ((\text{State}(\text{temp}) \wedge \langle \neg \rangle Z) \vee (\neg \text{State}(\text{temp}) \wedge t_j(\Psi))) v_2$$

Then there exists s'_2 s.t.

$$s_2 \Rightarrow_2 t_1 \Rightarrow_2 \dots \Rightarrow_2 t_n \Rightarrow_2 s'_2,$$

$\text{State}(\text{temp}) \in \text{abox}_2(t_i)$ for $i \in \{1, \dots, n\}$, $\text{State}(\text{temp}) \notin \text{abox}_2(s'_2)$, and $\mathcal{I}_2, s'_2 \models t_j(\Psi) v_2$. Since $s_1 \sim_J s_2$, there exists s'_1 , such that $s_1 \Rightarrow_1 s'_1$ and $s'_1 \sim_J s'_2$. Hence, by induction hypothesis, for every valuations v_1 that assign to each free variables x of Ψ a constant $c_1 \in \text{ADOM}(\text{abox}_1(s_1))$, such that $c_1 = c_2$ and $x/c_2 \in v_2$, we have $\mathcal{I}_1, s'_1 \models \Psi v_1$. Now, consider that $s_1 \Rightarrow_1 s'_1$, we then obtain that $\mathcal{I}_1, s_1 \models (\langle \neg \rangle \Psi) v_1$.

($\Phi = [\neg]\Psi$). The proof is similar to the case of $\Phi = \langle \neg \rangle \Psi$

($\Phi = \exists x. \Psi$). Assume that $\mathcal{I}_1, s_1 \models (\exists x. \Psi) v'_1$, where v'_1 is a valuation that assigns to each free variable of Ψ a constant $c_1 \in \text{ADOM}(\text{abox}_1(s_1))$. Then, by definition, there exists $c \in \text{ADOM}(\text{abox}_1(s_1))$ such that $\mathcal{I}_1, s_1 \models \Psi v_1$, where $v_1 = v'_1[x/c]$. By induction hypothesis, for every valuation v_2 that assigns to each free variable y of $t_j(\Psi)$ a constant $c_2 \in \text{ADOM}(\text{abox}_2(s_2))$, such that $c_1 = c_2$ and $y/c_1 \in v_1$, we have that $\mathcal{I}_2, s_2 \models t_j(\Psi) v_2$. Additionally, we have $v_2 = v'_2[x/c']$, where $c' \in \text{ADOM}(\text{abox}_2(s_2))$, and $c' = c$ because $\text{abox}_2(s_2) \simeq \text{abox}_1(s_1)$. Hence, we get $\mathcal{I}_2, s_2 \models (\exists x. t_j(\Psi)) v'_2$. Since $t_j(\exists x. \Phi) = \exists x. t_j(\Phi)$, thus we have $\mathcal{I}_2, s_2 \models t_j(\exists x. \Psi) v'_2$. The other direction can be shown similarly.

($\Phi = \forall x. \Psi$). The proof is similar to the case of $\Phi = \exists x. \Psi$.

Extension to arbitrary countable disjunction. Let Ψ be a countable set of $\mathcal{L}_A^{\text{EQL}}$ formulae. Given a transition system $\mathcal{T} = \langle \Delta, T, \Sigma, s_0, \text{abox}, \Rightarrow \rangle$, the semantics of $\bigvee \Psi$ is $(\bigvee \Psi)_v^{\mathcal{T}} = \bigcup_{\psi \in \Psi} (\psi)_v^{\mathcal{T}}$. Therefore, given a state $s \in \Sigma$ we have $\mathcal{T}, s \models (\bigvee \Psi) v$ if and only if there exists $\psi \in \Psi$ such that $\mathcal{T}, s \models \psi v$. Arbitrary countable conjunction can be obtained similarly.

Now, let $\mathcal{I}_1 = \langle \Delta, T, \Sigma_1, s_{01}, \text{abox}_1, \Rightarrow_1 \rangle$ and $\mathcal{I}_2 = \langle \Delta, T, \Sigma_2, s_{02}, \text{abox}_2, \Rightarrow_2 \rangle$. Consider two states $s_1 \in \Sigma_1$ and $s_2 \in \Sigma_2$ such that $s_1 \sim_J s_2$. By induction hypothesis, we have for every valuations v_1 and v_2 that assign to each of its free variables a constant $c_1 \in \text{ADOM}(\text{abox}_1(s_1))$ and $c_2 \in \text{ADOM}(\text{abox}_2(s_2))$, such that $c_2 = c_1$, we have that for every formula $\psi \in \Psi$, it holds $\mathcal{I}_1, s_1 \models \psi v_1$ if and only if $\mathcal{I}_2, s_2 \models t_j(\psi) v_2$. Given the semantics of $\bigvee \Psi$ above, this implies that $\mathcal{I}_1, s \models (\bigvee \Psi) v_1$ if and only if $\mathcal{I}_2, s \models (\bigvee t_j(\Psi)) v_2$, where $t_j(\Psi) = \{t_j(\psi) \mid \psi \in \Psi\}$. The proof is then obtained by observing that $\bigvee t_j(\Psi) = t_j(\bigvee \Psi)$.

Extension to full $\mu\mathcal{L}_A^{\text{EQL}}$. In order to extend the result to the whole $\mu\mathcal{L}_A^{\text{EQL}}$, we resort to the well-known result stating that fixpoints of the μ -calculus can be translated into the infinitary Hennessy Milner logic by iterating over *approximants*, where the approximant of index α is denoted by $\mu^\alpha Z.\Phi$ (resp. $\nu^\alpha Z.\Phi$) (cf. [170]). This is a standard result that also holds for $\mu\mathcal{L}_A^{\text{EQL}}$. In particular, approximants are built as follows:

$$\begin{aligned} \mu^0 Z.\Phi &= \text{false} & \nu^0 Z.\Phi &= \text{true} \\ \mu^{\beta+1} Z.\Phi &= \Phi[Z/\mu^\beta Z.\Phi] & \nu^{\beta+1} Z.\Phi &= \Phi[Z/\nu^\beta Z.\Phi] \\ \mu^\lambda Z.\Phi &= \bigvee_{\beta < \lambda} \mu^\beta Z.\Phi & \nu^\lambda Z.\Phi &= \bigwedge_{\beta < \lambda} \nu^\beta Z.\Phi \end{aligned}$$

where λ is a limit ordinal, and where fixpoints and their approximants are connected by the following properties: given a transition system \mathcal{T} and a state s of \mathcal{T}

- $s \in (\mu Z.\Phi)_{v,V}^{\mathcal{T}}$ if and only if there exists an ordinal α such that $s \in (\mu^\alpha Z.\Phi)_{v,V}^{\mathcal{T}}$ and, for every $\beta < \alpha$, it holds that $s \notin (\mu^\beta Z.\Phi)_{v,V}^{\mathcal{T}}$;
- $s \notin (\nu Z.\Phi)_{v,V}^{\mathcal{T}}$ if and only if there exists an ordinal α such that $s \notin (\nu^\alpha Z.\Phi)_{v,V}^{\mathcal{T}}$ and, for every $\beta < \alpha$, it holds that $s \in (\nu^\beta Z.\Phi)_{v,V}^{\mathcal{T}}$.

□

As a consequence, from Lemma 4.32 above, we can easily obtain the following lemma saying that two transition systems which are J-bisimilar can not be distinguished by any $\mu\mathcal{L}_A^{\text{EQL}}$ formula (in NNF) modulo a translation t_j .

Lemma 4.33. *Consider two transition systems \mathcal{T}_1 , and \mathcal{T}_2 such that $\mathcal{T}_1 \sim_J \mathcal{T}_2$. For every $\mu\mathcal{L}_A^{\text{EQL}}$ closed formula Φ (in NNF) we have:*

$$\mathcal{T}_1 \models \Phi \text{ if and only if } \mathcal{T}_2 \models t_j(\Phi).$$

Proof. Let $\mathcal{T}_1 = \langle \Delta, T, \Sigma_1, s_{01}, \text{abox}_1, \Rightarrow_1 \rangle$ and $\mathcal{T}_2 = \langle \Delta, T, \Sigma_2, s_{02}, \text{abox}_2, \Rightarrow_2 \rangle$. Since by the definition we have $s_{01} \sim_J s_{02}$, we obtain the proof as a consequence of Lemma 4.32 due to the fact that

$$\mathcal{T}_1, s_{01} \models \Phi \text{ if and only if } \mathcal{T}_2, s_{02} \models t_j(\Phi)$$

□

4.4.2 Transforming Standard GKABs into KABs

Here we explain how we transform S-GKABs into KABs with the aim of reducing verification of S-GKABs into KABs. As an important observation, notice that a program is essentially a specification of a sequence of action execution and the atomic step within a program execution is simply an action execution. Thus, the key idea of the translation is to inductively interpret a Golog program as a structure consisting of nested processes, suitably composed through the Golog operators. We mark the starting and ending point of each Golog subprogram, and use accessory assertions in the ABox to track states corresponding to subprograms. Each subprogram is then inductively translated into a set of actions and a set of condition-action rules encoding its entrance and termination conditions.

As the first step towards defining a generic translation to compile S-GKABs into KABs, we need to introduce the notion of sub-program and program IDs. The purpose

of the program IDs is to uniquely identify each sub-program of a Golog program. The motivation/intuition why we need to annotate a (sub-)program with IDs is as follows: Suppose we have a program

$$\delta = \mathbf{pick\ true}.\alpha_1(); \mathbf{pick\ true}.\alpha(); \mathbf{pick\ true}.\alpha_2(); \mathbf{pick\ true}.\alpha(); \mathbf{pick\ true}.\alpha_3()$$

which basically specifies a sequence of action execution $\alpha_1, \alpha, \alpha_2, \alpha, \alpha_3$. To translate δ into a set of KAB actions and a set of condition-action rules that “mimic the behavior” of δ , the idea is to enforce such sequence by using some reserved ABox assertions (*flags*) that act as pre and post condition of each action. Essentially, we aim to have that an action is executable when its pre-condition holds (when the corresponding ABox assertion is present in the KB) and after its execution, the action should make its post-condition holds (by adding the corresponding ABox assertion into the KB). However, since the action α occurs twice in two different position, we cannot use a single reserved ABox assertion for its pre or post condition. We need to differentiate the pre and post condition of the first and the second occurrence of α since they are different. In addition, later on we need to keep track and get the information about pre and post condition of each subprograms. Therefore, to differentiate those two different occurrences of α , we annotate them with IDs.

Sub-program **Definition 4.34** (Sub-program). Given a program δ , we define the notion of a *sub-program* of δ inductively as follows:

- δ is a sub-program of δ ,
- If δ is of the form $\delta_1 | \delta_2$, $\delta_1; \delta_2$, or **if** φ **then** δ_1 **else** δ_2 , then
 - δ_1 and δ_2 both are sub-programs of δ ,
 - each sub-program of δ_1 is a sub-program of δ ,
 - each sub-program of δ_2 is a sub-program of δ ,
- If δ is of the form **while** φ **do** δ_1
 - δ_1 is a sub-program of δ ,
 - each sub-program of δ_1 is a sub-program of δ ,

■

We say a program δ' *occurs in* δ if δ' is a sub-program of δ . Next, we define the notion of Golog programs with IDs, that are programs in which each of their sub-programs is annotated with a unique ID.

Golog Program with IDs **Definition 4.35** (Golog Program with IDs). Given a set of actions Γ , a *Golog program with ID* over Γ is an expression formed by the following grammar:

$$\begin{aligned} \langle id, \delta \rangle ::= & \langle id, \varepsilon \rangle \mid \langle id, \mathbf{pick\ } Q(\vec{p}).\alpha(\vec{p}) \rangle \mid \\ & \langle id, \langle id_1, \delta_1 \rangle | \langle id_2, \delta_2 \rangle \rangle \mid \langle id, \langle id_1, \delta_1 \rangle; \langle id_2, \delta_2 \rangle \rangle \mid \\ & \langle id, \mathbf{if\ } \varphi \mathbf{\ then\ } \langle id_1, \delta_1 \rangle \mathbf{\ else\ } \langle id_2, \delta_2 \rangle \rangle \mid \langle id, \mathbf{while\ } \varphi \mathbf{\ do\ } \langle id_1, \delta_1 \rangle \rangle \end{aligned}$$

where *id* is a *program ID* (that can be simply a string over some alphabets), and the rest of the things are the same as in Definition 4.1. ■

All notions related to Golog program can be defined similarly for the Golog program with IDs, since essentially we only annotate each sub-program with a unique ID. In the following we define a formal translation that transforms a Golog program into a Golog program with IDs. As a notation, given program IDs *id* and *id'*, we write *id.id'* to denote a string obtained by concatenating the strings *id* and *id'* consecutively.

Definition 4.36 (Program ID Assignment). We define a translation $\tau_{id}(\delta, id)$ that Program ID Assignment

1. takes a program δ as well as a program ID id , and
2. produces a Golog program with ID $\langle id, \delta_{id} \rangle$ such that each sub-program of δ is associated with a unique program ID and occurrence matters (i.e., for each sub-program δ' of δ such that δ' occurs more than once in δ , each of them has a different program ID)

formally as follows:

- $\tau_{id}(\varepsilon, id) = \langle id, \varepsilon \rangle$,
where id' is a fresh program ID.
- $\tau_{id}(\mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p}), id) = \langle id, \mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p}) \rangle$,
where id' is a fresh program ID.
- $\tau_{id}(\delta_1 | \delta_2, id) = \langle id, \tau_{id}(\delta_1, id.id') | \tau_{id}(\delta_2, id.id'') \rangle$,
where id' and id'' are fresh program IDs.
- $\tau_{id}(\delta_1; \delta_2, id) = \langle id, \tau_{id}(\delta_1, id.id'); \tau_{id}(\delta_2, id.id'') \rangle$,
where id' and id'' are fresh program IDs.
- $\tau_{id}(\mathbf{if} \ \varphi \ \mathbf{then} \ \delta_1 \ \mathbf{else} \ \delta_2, id) = \langle id, \mathbf{if} \ \varphi \ \mathbf{then} \ \tau_{id}(\delta_1, id.id') \ \mathbf{else} \ \tau_{id}(\delta_2, id.id'') \rangle$,
where id' and id'' are fresh program IDs.
- $\tau_{id}(\mathbf{while} \ \varphi \ \mathbf{do} \ \delta_1, id) = \langle id, \mathbf{while} \ \varphi \ \mathbf{do} \ \tau_{id}(\delta_1, id.id') \rangle$,
where id' is a fresh program ID.

Given a program δ , we say $\langle id, \delta_{id} \rangle$ is a program with IDs w.r.t. δ if $\tau_{id}(\delta, id) = \langle id, \delta_{id} \rangle$ where id is a fresh program ID and δ_{id} is a program with ID. ■

Definition 4.37 (Program ID Retrieval function). Let δ be a Golog program and $\langle id, \delta_{id} \rangle$ be its corresponding program with IDs w.r.t. δ , we define a function pid that Program ID Retrieval function

1. maps each sub-program of $\langle id, \delta \rangle$ into its unique ID. I.e., for each sub-program $\langle id', \delta' \rangle$ of $\langle id, \delta \rangle$, we have $pid(\langle id', \delta' \rangle) = id'$, and
2. additionally, for a technical reason related to the correctness proof of our translation from S-GKABs to KABs, for each action invocation $\langle id_\alpha, \mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p}) \rangle$ that is a sub-program of $\langle id, \delta_{id} \rangle$, we also have $pid(\langle id_\alpha.\varepsilon, \varepsilon \rangle) = id_\alpha.\varepsilon$ (where $id_\alpha.\varepsilon$ is a new ID simply obtained by concatenating id_α with a string ε).

■

For simplicity of the presentation, from now on we assume that every program is associated with ID. Note that every program without ID can be transform into a program with ID as above. Moreover we will not write the ID that is attached to a (sub-)program, and when it is clear from the context, we simply write $pid(\delta')$, instead of $pid(\langle id, \delta' \rangle)$, to denote the *unique program ID of a sub-program δ' of δ* that is based on its occurrence in δ .

We now proceed to define a translation t_G that, given a Golog program δ , produces a set of condition-action rules and a set of actions that mimic the behavior of δ . Note that the translation that we present below might not be the only way to translate a program into the set of condition-action rules and actions. Before we formally define t_G , in Example 4.38 we briefly illustrate the idea of how we translate a program and

also how we use the flags to make the resulting condition-action rules mimic the program behavior (Note that the purpose of Example 4.38 is only to give the ideas, and there are some simplification w.r.t. the translation t_G that we define later).

Example 4.38. Consider a program

$$\delta = \mathbf{pick} \ Q_1(\vec{x}).\alpha_1(\vec{x}); \mathbf{pick} \ Q_2(\vec{y}).\alpha_2(\vec{y})$$

and a state $s_0 = \langle A_0, m_0, \delta \rangle$. Suppose we have the following run:

$$\langle A_0, m_0, \delta \rangle \Rightarrow \langle A_1, m_1, \delta_1 \rangle \Rightarrow \langle A_2, m_2, \varepsilon \rangle$$

where $\delta_1 = \varepsilon; \mathbf{pick} \ Q(\vec{y}).\alpha_2(\vec{y})$. Notice that the execution of α_1 above change A_0 into A_1 and m_0 into m_1 (similarly for the execution of α_2). Now, to mimic the run above within KAB, we do the following:

1. we use three flags namely $\mathbf{Flag}(c_1)$, $\mathbf{Flag}(c_2)$, and $\mathbf{Flag}(c_3)$.
2. we translate α_1 into α'_1 such that

$$\mathbf{EFF}(\alpha'_1) = \mathbf{EFF}(\alpha_1) \cup \{\mathbf{true} \rightsquigarrow \mathbf{add} \ \{\mathbf{Flag}(c_2)\}, \mathbf{del} \ \{\mathbf{Flag}(c_1)\}\}$$

intuitively, the action α'_1 do the same thing as the action α_1 except that it deletes the ABox assertion $\mathbf{Flag}(c_1)$ and adds the ABox assertion $\mathbf{Flag}(c_2)$.

3. we translate α_2 into α'_2 such that

$$\mathbf{EFF}(\alpha'_2) = \mathbf{EFF}(\alpha_2) \cup \{\mathbf{true} \rightsquigarrow \mathbf{add} \ \{\mathbf{Flag}(c_3)\}, \mathbf{del} \ \{\mathbf{Flag}(c_2)\}\}$$

the intuition for α'_2 is similar to α'_1 .

4. we introduce the following condition-action rules:

- a) $Q_1(\vec{x}) \wedge \mathbf{Flag}(c_1) \mapsto \alpha'_1(\vec{x})$.
- b) $Q_2(\vec{y}) \wedge \mathbf{Flag}(c_2) \mapsto \alpha'_2(\vec{y})$.

intuitively, the first rule enforce that α_1 is executable when Q_1 is successfully evaluated over the current KB and $\mathbf{Flag}(c_1)$ is in the current ABox. Similarly, the second condition-action rule require that in order to have α'_2 executable, $\mathbf{Flag}(c_2)$ must be in the current ABox and Q_2 must be successfully evaluated. Since at the end of the execution of α'_1 it adds the assertion $\mathbf{Flag}(c_2)$, it is easy to see that the flags here enforce the sequence of action execution as in the specified program.

5. we add $\mathbf{Flag}(c_1)$ into the ABox in the state where the execution of δ begin. I.e., we have now the state $\langle A_0 \cup \{\mathbf{Flag}(c_1)\}, m_0 \rangle$. As an intuition, later on when we translate S-GKABs into KABs, we add the pre condition flag of the corresponding program of S-GKABs into the initial state of KABs.

Hence, we can now have the following run in KAB:

$$\langle A_0 \cup \{\mathbf{Flag}(c_1)\}, m_0 \rangle \Rightarrow \langle A_1 \cup \{\mathbf{Flag}(c_2)\}, m_1 \rangle \Rightarrow \langle A_2 \cup \{\mathbf{Flag}(c_3)\}, m_2 \rangle$$

where the run above is obtained by sequentially executing α'_1 and α'_2 .

Roughly speaking, $\text{Flag}(c_1)$ (resp. $\text{Flag}(c_2)$) is the pre (resp. post) condition of the program **pick** $Q_1(\vec{x}).\alpha_1(\vec{x})$. Generalizing the idea, we can say that $\text{Flag}(c_1)$ (resp. $\text{Flag}(c_3)$) is the pre (resp. post) condition of the program δ . Thus, later on we will see that the translation t_G will translate a program δ recursively over the sub-programs δ' of δ and each sub-program has their own pre/post condition.

As learned from Example 4.38, the important intuition to create the translation t_G is that a program is essentially specifying a particular sequence of action execution. Now, generalizing the idea in Example 4.38, the translation t_G basically takes as inputs:

- a Golog program δ ,
- two flags (i.e., special ABox assertions) st and ed .

and produces the corresponding set of condition-action rules Π and actions Γ that mimic the execution of the program δ . Moreover, the flags st and ed are used by Π and Γ to mark the start and end of a run τ in KAB that is “induced by” Π (together with Γ) and mimics the run τ' that is “induced by” δ in S-GKAB. The translation t_G works recursively over the structure of the given program δ , while at the same time for each sub-program δ' of δ , it produces and accumulates the corresponding set of actions and condition-action rules for δ' in order to produce the whole set of actions Γ and condition-action rules Π that mimic the given program δ . Formally, the translation t_G is defined as follows:

Definition 4.39 (Program Translation). A program translation t_G that takes as Program Translation
inputs:

- (i) A program δ over a set of actions Γ ,
- (ii) Two flags (i.e., two special ABox assertions which will be used as markers indicating the start and the end of the execution of a program δ).

and produces as outputs:

- (i) pre is a function that maps the ID of δ (as well as the IDs of all sub-programs of δ) to the flag (called *start flag*) that acts as a marker indicating the start of the run induced by the corresponding set of condition-action rules and actions obtained from δ (as well as all of its sub-programs),
- (ii) $post$ is a function that maps the ID of δ (as well as the IDs of all sub-programs of δ) to the flag (called *end flag*) that acts as a marker indicating the end of the run induced by the corresponding set of condition-action rules and actions obtained from δ (as well as all of its sub-programs),
- (iii) Π is a process (a set of condition-action rules),
- (iv) Γ' is a set of actions.

I.e., $t_G(st, \delta, ed) = \langle pre, post, \Pi, \Gamma' \rangle$, where st and ed are flags. Formally, $t_G(st, \delta, ed)$ is inductively defined over the structure of a program δ as follows:

1. For the case of $\delta = \varepsilon$ (i.e., δ is an empty program):

$$t_G(st, \varepsilon, ed) = \langle pre, post, \{st \mapsto \alpha_\varepsilon()\}, \{\alpha_\varepsilon\} \rangle,$$

where

- $pre = \{[pid(\varepsilon) \rightarrow st]\}$,
 - $post = \{[pid(\varepsilon) \rightarrow ed]\}$,
 - α_ε is of the form $\alpha_\varepsilon() : \{\text{true} \rightsquigarrow \mathbf{add} \{ed, \text{State}(temp)\}, \mathbf{del} \{st\}\}$;
2. For the case of $\delta = \mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p})$ (i.e., δ is an action invocation) with $pid(\mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p})) = id_\alpha$:

$$t_{\mathcal{G}}(st, \mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p}), ed) = \langle pre, post, \Pi, \Gamma' \rangle,$$

where

- $pre = \{[pid(\mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p})) \rightarrow st]\} \cup pre'$,
 - $post = \{[pid(\mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p})) \rightarrow ed]\} \cup post'$,
 - $\Pi = \{Q(\vec{p}) \wedge st \mapsto \alpha'(\vec{p})\} \cup \Pi'$,
 - $\Gamma' = \{\alpha'\} \cup \Gamma''$, where

$$\begin{aligned} \text{EFF}(\alpha') = & \text{EFF}(\alpha) \cup \\ & \{\text{true} \rightsquigarrow \mathbf{add} \{ed\}, \mathbf{del} \{st, \text{State}(temp)\}\} \cup \\ & \{\text{Noop}(x) \rightsquigarrow \mathbf{del} \ \text{Noop}(x)\}, \end{aligned}$$
 - $t_{\mathcal{G}}(ed, \varepsilon, ed) = \langle pre', post', \Pi', \Gamma'' \rangle$, where $pid(\varepsilon) = id_{\alpha.\varepsilon}$
3. For the case of $\delta = \delta_1 | \delta_2$ (i.e., δ is a non-deterministic choice between programs):

$$t_{\mathcal{G}}(st, \delta_1 | \delta_2, ed) = \langle pre, post, \Pi, \Gamma \rangle,$$

where

- $\Pi = \{st \mapsto \gamma_{\delta_1}(), st \mapsto \gamma_{\delta_2}()\} \cup \Pi_1 \cup \Pi_2$,
 - $\Gamma = \Gamma_1 \cup \Gamma_2 \cup \{\gamma_{\delta_1}, \gamma_{\delta_2}\}$, where
 - $\gamma_{\delta_1}() : \{\text{true} \rightsquigarrow \mathbf{add} \{\text{Flag}(c_1), \text{State}(temp)\}, \mathbf{del} \{st\}\}$,
 - $\gamma_{\delta_2}() : \{\text{true} \rightsquigarrow \mathbf{add} \{\text{Flag}(c_2), \text{State}(temp)\}, \mathbf{del} \{st\}\}$,
 - $pre = \{[pid(\delta_1 | \delta_2) \rightarrow st]\} \cup pre_1 \cup pre_2$,
 - $post = \{[pid(\delta_1 | \delta_2) \rightarrow ed]\} \cup post_1 \cup post_2$,
 - $t_{\mathcal{G}}(\text{Flag}(c_1), \delta_1, ed) = \langle pre_1, post_1, \Pi_1, \Gamma_1 \rangle$,
 - $t_{\mathcal{G}}(\text{Flag}(c_2), \delta_2, ed) = \langle pre_2, post_2, \Pi_2, \Gamma_2 \rangle$,
 - $c_1, c_2 \in \Delta_0$ are fresh constants;
4. $t_{\mathcal{G}}(st, \delta_1; \delta_2, ed) = \langle pre, post, \Pi_1 \cup \Pi_2, \Gamma_1 \cup \Gamma_2 \rangle$, where
- $pre = \{[pid(\delta_1; \delta_2) \rightarrow st]\} \cup pre_1 \cup pre_2$,
 - $post = \{[pid(\delta_1; \delta_2) \rightarrow ed]\} \cup post_1 \cup post_2$,
 - $t_{\mathcal{G}}(st, \delta_1, \text{Flag}(c)) = \langle pre_1, post_1, \Pi_1, \Gamma_1 \rangle$,
 - $t_{\mathcal{G}}(\text{Flag}(c), \delta_2, ed) = \langle pre_2, post_2, \Pi_2, \Gamma_2 \rangle$,
 - $c \in \Delta_0$ is a fresh constant;

5. $t_{\mathcal{G}}(st, \mathbf{if} \ \varphi \ \mathbf{then} \ \delta_1 \ \mathbf{else} \ \delta_2, ed) = \langle pre, post, \Pi, \Gamma \rangle$, where

- $pre = \{[pid(\mathbf{if} \ \varphi \ \mathbf{then} \ \delta_1 \ \mathbf{else} \ \delta_2) \rightarrow st]\} \cup pre_1 \cup pre_2$,
- $post = \{[pid(\mathbf{if} \ \varphi \ \mathbf{then} \ \delta_1 \ \mathbf{else} \ \delta_2) \rightarrow ed]\} \cup post_1 \cup post_2$,
- $\Pi = \{st \wedge \varphi \mapsto \gamma_{if}(), st \wedge \neg \varphi \mapsto \gamma_{else}()\} \cup \Pi_1 \cup \Pi_2$,
- $\Gamma = \Gamma_1 \cup \Gamma_2 \cup \{\gamma_{if}, \gamma_{else}\}$, where
 - $\gamma_{if}() : \{\text{true} \rightsquigarrow \mathbf{add} \{\text{Flag}(c_1), \text{State}(temp)\}, \mathbf{del} \{st\}\}$,
 - $\gamma_{else}() : \{\text{true} \rightsquigarrow \mathbf{add} \{\text{Flag}(c_2), \text{State}(temp)\}, \mathbf{del} \{st\}\}$,

- $t_{\mathcal{G}}(\text{Flag}(c_1), \delta_1, ed) = \langle pre_1, post_1, \Pi_1, \Gamma_1 \rangle$,
 - $t_{\mathcal{G}}(\text{Flag}(c_2), \delta_2, ed) = \langle pre_2, post_2, \Pi_2, \Gamma_2 \rangle$,
 - $c_1, c_2 \in \Delta_0$ are fresh constants;
6. $t_{\mathcal{G}}(st, \text{while } \varphi \text{ do } \delta, ed) = \langle pre, post, \Pi, \Gamma \rangle$, where
- $pre = \{[pid(\text{while } \varphi \text{ do } \delta) \rightarrow st]\} \cup pre'$
 - $post = \{[pid(\text{while } \varphi \text{ do } \delta) \rightarrow ed]\} \cup post'$
 - $\Pi = \Pi' \cup \Pi_{loop}$, where Π_{loop} contains:
 - $st \wedge \varphi \wedge \neg \text{Noop}(noop) \mapsto \gamma_{doLoop}()$,
 - $st \wedge (\neg \varphi \vee \text{Noop}(noop)) \mapsto \gamma_{endLoop}()$,
 - $\Gamma = \Gamma' \cup \Gamma_{loop}$, where Γ_{loop} contains the following:
 - $\gamma_{doLoop}() : \{\text{true} \rightsquigarrow$
 $\text{add } \{\text{Flag}(lStart), \text{Noop}(noop), \text{State}(temp)\}, \text{del } \{st\}\},$
 - $\gamma_{endLoop}() : \{\text{true} \rightsquigarrow$
 $\text{add } \{ed, \text{State}(temp)\}, \text{del } \{st, \text{Noop}(noop)\}\},$
 - $t_{\mathcal{G}}(\text{Flag}(lStart), \delta, st) = \langle pre', post', \Pi', \Gamma' \rangle$,
 - $noop, lStart \in \Delta_0$ are fresh constants.

■

For compactness reason, in the following we often simply write $pre(\delta)$ to abbreviate the notation $pre(pid(\delta))$ that essentially returns the start flag of a program with program ID $pid(\delta)$. Similarly for $post(\delta)$. To give the intuition of the translation $\tau_{\mathcal{G}}$ as well as to show some of its properties, we first introduce the notion when a state of an S-GKAB is mimicked by a state of a KAB. Roughly speaking, a state $\langle A_g, m_g, \delta_g \rangle$ of an S-GKAB is mimicked by a state $\langle A_k, m_k \rangle$ of a KAB if

1. The corresponding ABox of those states contain the same assertions except the special markers. I.e., they are equal modulo special markers (cf. Definition 4.27),
2. They both have the same service call map, and
3. The ABox A_k contains the start flag of the program δ_g .

This notion is formally defined as follows.

Definition 4.40. Let $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ be an S-GKAB with transition system $\Upsilon_{\mathcal{G}}^{fs}$, Mimicked States and $\tau_{\mathcal{G}}(\mathcal{G}) = \langle T, A'_0, \Gamma', \Pi' \rangle$ be a KAB with transition system $\Upsilon_{\tau_{\mathcal{G}}(\mathcal{G})}$ obtained from \mathcal{G} through $\tau_{\mathcal{G}}$. Consider two states $\langle A_g, m_g, \delta_g \rangle$ of $\Upsilon_{\mathcal{G}}^{fs}$ and $\langle A_k, m_k \rangle$ of $\Upsilon_{\tau_{\mathcal{G}}(\mathcal{G})}$. We say $\langle A_g, m_g, \delta_g \rangle$ is mimicked by $\langle A_k, m_k \rangle$ (or equivalently $\langle A_k, m_k \rangle$ mimics $\langle A_g, m_g, \delta_g \rangle$), written $\langle A_g, m_g, \delta_g \rangle \cong \langle A_k, m_k \rangle$, if

1. $A_k \simeq A_g$,
2. $m_k = m_g$, and
3. $pre(\delta_g) \in A_k$.

■

The intuition of each recursive step in the translation $t_{\mathcal{G}}$ is then explained as follows:

- For the case of $\delta = \varepsilon$, given a program ε , a start flag st , and an end flag ed , the translation $t_{\mathcal{G}}$ do the following:
 1. it produces a function pre such that $pre(pid(\varepsilon)) = st$,
 2. it produces a function $post$ such that $post(pid(\varepsilon)) = ed$,

3. it produces a singleton set of actions containing an action α_ε that adds (resp. deletes) the end flag ed (resp. the start flag st).
4. Additionally, α_ε also adds the assertion $\mathbf{State}(temp)$. The idea is to make the state generated by this action ignored (during the verification). Because in S-GKABs, when we have a program ε , they essentially do not make any transition and the program execution can be considered completed (cf. Definition 4.14). Basically, it is also the reason why we introduce the translation t_j (cf. Definition 4.31) that bypass some intermediate states (states containing $\mathbf{State}(temp)$).

As a further intuition for the translation of the case $\delta = \varepsilon$, consider a program

$$\delta = \mathbf{pick\ true}.\alpha_1(); \delta_2; \mathbf{pick\ true}.\alpha_3(),$$

where $\delta_2 = \mathbf{pick\ true}.\alpha_2() \mid \varepsilon$. Thus, due to the use of non-deterministic choice construct, there are two possible sequences of action execution namely:

- (1) α_1 and then followed by α_2 and α_3 .
- (2) α_1 and then followed by α_3 .

To emulate such situation in KAB,

- for (1), the post-condition of α_1 should be the pre-condition for α_2 ,
- for (2), the post-condition of α_1 should be the pre-condition for α_3 .

Hence, when we translate α_1 , it is not clear which end flag that α_1 should add at the end of its execution. Therefore, it is one of the reason why we translate ε into an action α_ε that only change the flag. Note that it is also aligned with the general intuition of the translation t_G in which for each sub-program there will be a corresponding start flag and end flag. Thus, for the case (2) above, we then have an action α_ε that bridges the execution of α_1 and α_3 . Essentially, the translation t_G will translate α_1 such that at the end of its execution it adds an end flag that is also the start flag of δ_2 and then, no matter which choice that is taken in δ_2 , at the end of execution of δ_2 , it will add an end flag that is also the start flag of α_3 (i.e., in the case of choosing ε , the action α_ε will put the start flag of α_3 so that α_3 can be fired, however the generated state by α_ε will be marked as an intermediate state by the assertion $\mathbf{State}(temp)$ and hence it will be ignored during the verification). Similarly for the case of a program sequence, as an intuition, consider the program

$$\delta = \varepsilon; \mathbf{pick\ true}.\alpha_1(); \mathbf{pick\ true}.\alpha_2().$$

Let A be an ABox and m be a service call map, then we have

$$\langle A, m, \delta \rangle \xrightarrow{\alpha_1\sigma, f_S} \langle A', m', \delta' \rangle$$

where $\delta' = \mathbf{pick\ true}.\alpha_2()$. However, following the general idea of translation t_G , especially for the case of program sequence $\delta = \delta_1; \delta_2$, each sub-program will be translated into a set of condition-action rules and actions, and each sub-program will have the corresponding start (resp. end) flag that will drive the execution of actions in KAB. In the case of $\delta = \delta_1; \delta_2$, the end flag of δ_1 should be the start flag of δ_2 . Therefore, when δ_1 is ε , we need to change the flag from the start flag of δ_1 into the end flag of δ_1 . This is also one of the reason why we translate ε this way.

- For the case of $\delta = \mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p})$, given a program $\mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p})$, a start flag st , and an end flag ed , the idea of translation t_G for this case is as follows:
 - the translation t_G produces a function pre (resp. $post$) such that $pre(pid(\mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p}))) = st$ (resp. $post(pid(\mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p}))) = ed$),
 - we translate the action α into an action α' such that α' does the same thing as α except that it also does the following:
 - * α' adds the end flag ed and deletes the start flag st .
 - * α' deletes all flags made by the concept name **Noop**. This deletion is related to the translation of the while loop construct. In the semantics of while loop, as it is explained in the beginning of this chapter, for any ABox A and service call map m , we have that $\langle A, m, \mathbf{while} \ \varphi \ \mathbf{do} \ \delta' \rangle \in \mathbb{F}$ if $\text{ASK}(\varphi, T, A) = \text{true}$, and $\langle A, m, \delta' \rangle \in \mathbb{F}$. I.e., when δ' is considered to be completed, the whole loop can be considered to be completed as well. To check this, we make use the flags made by the concept name **Noop**. The idea is that if there is still a flag made by **Noop** at the end of the loop, then it means that there is no action that is executed. Therefore, here we translate an action such that it clears all flags made by **Noop** to give a sign that there is an action that is executed.
 - we create a condition-action rule $Q(\vec{p}) \wedge st \mapsto \alpha'(\vec{p})$. So, the idea is that α' can be executed when st is in the current ABox and at the end of the execution of α' , we have ed in the ABox.
 - we also recursively call the translation $t_G(ed, \varepsilon, ed)$ where $pid(\varepsilon) = id_\alpha.\varepsilon$. This step is needed for some part of the correctness proof of the translation t_G . In particular, later on we will show that given a GKAB state $s_1 = \langle A_g, m_g, \delta_g \rangle$ and a KAB state $s_2 = \langle A_k, m_k \rangle$ such that A_k mimics A_g , if s_1 can reach a state $s'_1 = \langle A'_g, m'_g, \delta'_g \rangle$, then there exists a state $s'_2 = \langle A'_k, m'_k \rangle$ such that s_2 can reach s'_2 and s'_2 mimics s'_1 . Thus, for the base case of the proof (the case of action invocation), since we have

$$\langle A_g, m_g, \mathbf{pick} \ Q(\vec{x}).\alpha(\vec{x}) \rangle \xrightarrow{\alpha\sigma, fs} \langle A'_g, m'_g, \varepsilon \rangle,$$

then we need to have the start flag of ε in A'_k . Therefore we need to translate ε (where $pid(\varepsilon) = id_\alpha.\varepsilon$) such that it has the start flag ed (that is also the end flag of α). In addition, since we don't need to change further the flag, we call the translation t_G for ε with the end flag ed .

- For the case of $\delta = \delta_1; \delta_2$, given a program $\delta_1; \delta_2$, a start flag st , and an end flag ed , the translation t_G do the following:
 - it introduces a fresh flag $\mathbf{Flag}(c)$ that become the end flag of δ_1 and also the start flag of δ_2 . The intuition is that $\mathbf{Flag}(c)$ bridges/connects the run in the KAB that emulates δ_1 and the run in the KAB that emulates δ_2 .
 - it recursively translates δ_1 and δ_2 using the translation t_G . For translating δ_1 , the translation t_G is fired with the start flag st and the end flag $\mathbf{Flag}(c)$, while for translating δ_2 , the translation t_G is fired with the start flag $\mathbf{Flag}(c)$ and the end flag ed . The idea is to enforce that δ_2 will be executed after δ_1 has been successfully executed.

- it produces a set of condition-action rules (resp. a set of actions) that is obtained by merging the sets of condition-action rules (resp. the sets of actions) that are obtained from recursively translating δ_1 and δ_2
 - it produces a function pre such that $pre = \{[pid(\delta_1; \delta_2) \rightarrow st]\} \cup pre_1 \cup pre_2$, where pre_1 (resp. pre_2) is obtained from the translation of δ_1 (resp. δ_2). Intuitively, pre is obtained by mapping the ID of the current program into the given start flag st while also accumulating the mapping between IDs and start flags obtained from the translation of all of its sub-program.
 - the intuition for $post$ is similar to pre .
- For the case of $\delta = \delta_1 | \delta_2$, given a program $\delta_1 | \delta_2$, a start flag st and an end flag ed , the translation t_G do the following:
 - it introduces two flags $Flag(c_1)$ and $Flag(c_2)$. One for the start flag of δ_1 and one for the start flag of δ_2 . Two corresponding actions also generated in order to put either $Flag(c_1)$ or $Flag(c_2)$ into the Abox (i.e., to mark the start of the run in the KAB that emulates the run in the S-GKAB that is induced by either δ_1 or δ_2).
 - it recursively translates δ_1 with the start flag $Flag(c_1)$ and the end flag ed . Similarly for δ_2 except that the start flag is $Flag(c_2)$
 - it constructs pre , $post$, Π , and Γ by accumulating the results from translating δ_1 and δ_2 (similar to the previous construct).

Furthermore, the reason why we need to introduce two fresh flags for the start flag of δ_1 and δ_2 is to enforce that each sub-program has a unique start flag. Later on, we will see that it is important because as a step to show the correctness of the translation t_G , we show that the behavior of the transition system will be the “same” starting from a GKAB state $s_g = \langle A_g, m_g, \delta_1 | \delta_2 \rangle$ and a KAB state $s_k = \langle A_k, m_k \rangle$ in which s_k mimics s_g . Now, suppose that we do not invent two fresh flags for the translation of δ_1 as well as δ_2 , and we simply use st as their start flag, then we will have that $\delta_1 | \delta_2$, δ_1 , and δ_2 are having the same start flag. Hence it is easy to see that the property that we want to show above cannot be proven. As an intuition, suppose that the execution of δ_1 will be stuck while the execution of δ_2 is not. Then we cannot say that the behavior of the transition system will be the “same” starting from $s_1 = \langle A_g, m_g, \delta_1 \rangle$ and $s_2 = \langle A_k, m_k \rangle$ where s_2 mimics s_1 . Because $\langle A_g, m_g, \delta_1 \rangle$ will be stuck but $\langle A_k, m_k \rangle$ will be able to continue the execution by emulating the execution of δ_2 and this is possible since in any case the start flag of δ_2 is the same as the start flag of δ_1 and A_k contains such start flag.

- For the case of $\delta = \mathbf{if} \varphi \mathbf{then} \delta_1 \mathbf{else} \delta_2$, the idea is as follows:
 - we first introduce two fresh flags that will be used as the start flags of δ_1 and δ_2 .
 - to check whether φ is successfully evaluated over the current KB or not, we introduce two condition-action rules as follows:
 1. $st \wedge \varphi \mapsto \gamma_{if}()$,
 2. $st \wedge \neg \varphi \mapsto \gamma_{else}()$
 in case φ is successfully evaluated, the condition of the first rule will be satisfied, and then the action γ_{if} will be fired. As a consequence, γ_{if} will add

- the start flag for δ_1 . In case φ is not successfully evaluated, the condition of the second rule will be satisfied, and then the action γ_{else} will be fired. As a consequence, γ_{else} will add the start flag for δ_2 .
- similar to the other previous construct, in this case the translation t_g also recursively translates the program δ_1 and δ_2 each with their own start flag but both of them have the same end flag.
 - note that the states generated by either γ_{if} or γ_{else} will be just intermediate states that will be ignored during the verification.
 - the idea for the construction of pre , $post$, Π , and Γ , that involves accumulating the results from translating δ_1 and δ_2 , is similar to the other previous construct.
- For the case of $\delta = \mathbf{while} \ \varphi \ \mathbf{do} \ \delta'$, the idea is as follows:
 - when we translate a loop, we introduce a fresh flag $\mathbf{Flag}(lStart)$ that is used to mark the situation when we enter the body of the corresponding loop.
 - we introduce an action γ_{doLoop} that adds the flag $\mathbf{Flag}(lStart)$.
 - based on the semantics of the loop construct, as it is explained in the beginning of this chapter, for any ABox A and service call map m , we have that $\langle A, m, \mathbf{while} \ \varphi \ \mathbf{do} \ \delta' \rangle \in \mathbb{F}$ if $\mathbf{ASK}(\varphi, T, A) = \mathbf{true}$, and $\langle A, m, \delta' \rangle \in \mathbb{F}$. I.e., when δ' is considered to be completed, the whole loop can be considered to be completed. To mimic such situation in KABs, when we translate a loop, we introduce a fresh flag $\mathbf{Noop}(noop)$ that will be also added by γ_{doLoop} . The idea is that the flag $\mathbf{Noop}(noop)$ should be added when we start mimicking the computation of the loop in KABs and it should be deleted in case there is an action that is executed within the run that emulates the body of the loop. Otherwise, if the flag $\mathbf{Noop}(noop)$ remains in the ABox until the end of the run that mimics the loop computation, then it means that there is no action execution (Recall that we translate an action such that it will delete all flags made by \mathbf{Noop}).
 - the program δ' is recursively translated by t_g with the start flag $\mathbf{Flag}(lStart)$ and the end flag st . The reason of using st as the end flag of δ' is to enforce repetition. So, when the corresponding run in KAB have finished mimicking the execution of δ' , it should check whether the guard of the loop still holds or not. In case yes, it should emulate δ' again. Hence, by having st as the end flag, the execution will be back to the beginning again and it will check the guard of the loop again.
 - we also introduce a condition-action rule

$$st \wedge \varphi \wedge \neg \mathbf{Noop}(noop) \mapsto \gamma_{doLoop}()$$

that basically guards the execution of γ_{doLoop} such that γ_{doLoop} will be executed when the following hold:

1. when we start to (re-)enter the loop (marked by st),
2. when the guard of the loop φ is satisfied, and
3. when there is no assertion $\mathbf{Noop}(noop)$.

The reason why we need to check the presence of $\mathbf{Noop}(noop)$ is because we want to check whether there was any action execution within the body of

the loop or not (Notice that this condition-action rule will be re-evaluated at the end of a loop).

- we also introduce an action $\gamma_{endLoop}$ that adds the flag ed and makes us leave the loop. The execution of $\gamma_{endLoop}$ is then guarded by the condition-action rule

$$st \wedge (\neg\varphi \vee \mathbf{Noop}(noop)) \mapsto \gamma_{endLoop}()$$

in which its left hand side will be checked when we (re-)enter the loop. This rule says that $\gamma_{endLoop}()$ will be fired when the guard of the loop φ is not satisfied or when there is $\mathbf{Noop}(noop)$ (i.e., there is no need to re-execute the loop).

- the idea for the construction of pre , $post$, Π , and Γ that involves accumulating the results from translating δ' is similar to the other previous construct.

Below we show an important Lemma about the function pre and $post$ that will be used quite often later when we reduce the verification of S-GKABs into KABs.

Lemma 4.41. *Given a program δ over a set Γ of actions. We have*

$$t_{\mathcal{G}}(st, \delta, ed) = \langle pre, post, \Pi, \Gamma \rangle \text{ if and only if } pre(\delta) = st \text{ and } post(\delta) = ed$$

Proof. Directly follows from the definition of $t_{\mathcal{G}}$. □

Having $t_{\mathcal{G}}$ in hand, we define a translation $\tau_{\mathcal{G}}$ that compile S-GKABs into KABs as follows.

*Translation from
S-GKABs to KABs*

Definition 4.42 (Translation from S-GKABs to KABs). We define a translation $\tau_{\mathcal{G}}$ that takes an S-GKAB $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ as the input and produces a KAB $\tau_{\mathcal{G}}(\mathcal{G}) = \langle T, A'_0, \Gamma', \Pi' \rangle$ s.t.

- $A'_0 = A_0 \cup \{\mathbf{Flag}(start)\}$, and
 - $t_{\mathcal{G}}(\mathbf{Flag}(start), \delta, \mathbf{Flag}(end)) = \langle pre, post, \Pi', \Gamma' \rangle$.
-

Next, we define the notion of temp adder/deleter action as follows.

*Temp Marker Adder
Action*

Definition 4.43 (Temp Marker Adder Action). Let $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ be an S-GKAB and $\tau_{\mathcal{G}}(\mathcal{G}) = \langle T, A'_0, \Gamma', \Pi' \rangle$ be the corresponding KAB obtained from \mathcal{G} via $\tau_{\mathcal{G}}$. An action $\alpha \in \Gamma$ is a *temp adder action* of $\tau_{\mathcal{G}}(\mathcal{G})$ if there exists an effect $e \in \mathbf{EFF}(\alpha)$ of the form $[q^+] \wedge Q^- \rightsquigarrow \mathbf{add} F^+, \mathbf{del} F^-$ such that $\mathbf{State}(temp) \in F^+$. We write Γ_{ϵ}^+ to denote the set of temp adder actions of $\tau_{\mathcal{G}}(\mathcal{G})$. ■

*Temp Marker Deleter
Action*

Definition 4.44 (Temp Marker Deleter Action). Let $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ be an S-GKAB and $\tau_{\mathcal{G}}(\mathcal{G}) = \langle T, A'_0, \Gamma', \Pi' \rangle$ be the corresponding KAB obtained from \mathcal{G} via $\tau_{\mathcal{G}}$. An action $\alpha \in \Gamma$ is a *temp deleter action* of $\tau_{\mathcal{G}}(\mathcal{G})$ if there exists an effect $e \in \mathbf{EFF}(\alpha)$ of the form $[q^+] \wedge Q^- \rightsquigarrow \mathbf{add} F^+, \mathbf{del} F^-$ such that $\mathbf{State}(temp) \in F^-$. We write Γ_{ϵ}^- to denote the set of temp deleter actions of $\tau_{\mathcal{G}}(\mathcal{G})$. ■

Roughly speaking, a temp adder action is an action that adds the ABox assertion $\mathbf{State}(temp)$. Similarly, a temp deleter action is an action that removes the ABox assertion $\mathbf{State}(temp)$. In the following, we show several important properties of temp adder/deleter action that will be used later for reducing the verification of S-GKABs into KABs.

Lemma 4.45. *Let \mathcal{G} be an S-GKAB, $\tau_{\mathcal{G}}(\mathcal{G}) = \langle T, A'_0, \Gamma', \Pi' \rangle$ be the corresponding KAB obtained from \mathcal{G} via $\tau_{\mathcal{G}}$, and Γ_{ε}^+ (resp. Γ_{ε}^-) be a set of temp adder (resp. deleter) actions of $\tau_{\mathcal{G}}(\mathcal{G})$. We have that $\Gamma' = \Gamma_{\varepsilon}^+ \uplus \Gamma_{\varepsilon}^-$.*

Proof. Trivially true by observing Definitions 4.39, 4.43 and 4.44. \square

Lemma 4.46. *Let \mathcal{G} be an S-GKAB, $\tau_{\mathcal{G}}(\mathcal{G}) = \langle T, A'_0, \Gamma', \Pi' \rangle$ be the corresponding KAB (with transition system $\Upsilon_{\tau_{\mathcal{G}}(\mathcal{G})}$) obtained from \mathcal{G} via $\tau_{\mathcal{G}}$, and Γ_{ε}^+ be a set of temp adder actions of $\tau_{\mathcal{G}}(\mathcal{G})$. Consider a state $\langle A_k, m_k \rangle$ of $\Upsilon_{\tau_{\mathcal{G}}(\mathcal{G})}$, if there exists a state $\langle A'_k, m'_k \rangle$ such that $\langle A_k, m_k \rangle \xrightarrow{\alpha\sigma} \langle A'_k, m'_k \rangle$, and $\text{State}(\text{temp}) \in A'_k$ then σ is an empty substitution, $\alpha \in \Gamma_{\varepsilon}^+$, α does not involve any service calls, $A'_k \simeq A_k$ and $m'_k = m_k$.*

Proof. Since $\text{State}(\text{temp}) \in A'_k$, then by Definition 4.43 and Lemma 4.45 we must have $\alpha \in \Gamma_{\varepsilon}^+$. By the definition of translation $t_{\mathcal{G}}$ (see Definition 4.39), any actions in Γ_{ε}^+ does not involve service calls and only do a manipulation on special markers. Thus, it is easy to see that $A'_k \simeq A_k$ and $m'_k = m_k$. \square

The following lemma basically says that if there is a transition from a state s to s' that is obtained by execution an action α' and $\text{State}(\text{temp}) \notin \text{abox}(s'_k)$, then the action α' must be an action that deletes $\text{State}(\text{temp})$ and it must be obtained from a translation of an atomic action invocation.

Lemma 4.47. *Let $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ be an S-GKAB, $\tau_{\mathcal{G}}(\mathcal{G}) = \langle T, A'_0, \Gamma', \Pi' \rangle$ be the corresponding KAB (with transition system $\Upsilon_{\tau_{\mathcal{G}}(\mathcal{G})}$) obtained from \mathcal{G} via $\tau_{\mathcal{G}}$, and Γ_{ε}^+ be a set of temp adder actions of $\tau_{\mathcal{G}}(\mathcal{G})$. Consider a state $\langle A_k, m_k \rangle$ of $\Upsilon_{\tau_{\mathcal{G}}(\mathcal{G})}$, if there exists a state $\langle A'_k, m'_k \rangle$ such that $\langle A_k, m_k \rangle \xrightarrow{\alpha'\sigma} \langle A'_k, m'_k \rangle$, and $\text{State}(\text{temp}) \notin A'_k$ then $\alpha' \in \Gamma_{\varepsilon}^-$, and there exists action invocation **pick** $Q(\vec{p}).\alpha(\vec{p})$ in the sub-program of δ such that α' is obtained from the translation of **pick** $Q(\vec{p}).\alpha(\vec{p})$ via $t_{\mathcal{G}}$.*

Proof. Since $\text{State}(\text{temp}) \notin A'_k$, then by Definition 4.44 and Lemma 4.45 we must have $\alpha' \in \Gamma_{\varepsilon}^-$. By the definition of translation $t_{\mathcal{G}}$ (see Definition 4.39), α' must be obtained from the translation of an action invocation **pick** $Q(\vec{p}).\alpha(\vec{p})$ in the sub-program of δ . \square

The following lemma shows that given two action invocations that has different program ID, we have that their start flags are different. I.e., any actions invocations that occur in a different place inside a certain program will have different start flag. This claim is formalized below.

Lemma 4.48. *Let $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ be an S-GKAB, $\tau_{\mathcal{G}}(\mathcal{G}) = \langle T, A'_0, \Gamma', \Pi' \rangle$ be the corresponding KAB (with transition system $\Upsilon_{\tau_{\mathcal{G}}(\mathcal{G})}$) obtained from \mathcal{G} via $\tau_{\mathcal{G}}$, and Γ_{ε}^+ be a set of temp adder actions of $\tau_{\mathcal{G}}(\mathcal{G})$. Consider two action invocations **pick** $Q_1(\vec{x}).\alpha_1(\vec{x})$ and **pick** $Q_2(\vec{y}).\alpha_2(\vec{y})$ that are sub-programs of δ . We have that $\text{pid}(\text{pick } Q_1(\vec{x}).\alpha_1(\vec{x})) \neq \text{pid}(\text{pick } Q_2(\vec{y}).\alpha_2(\vec{y}))$ if and only if $\text{pre}(\text{pid}(\text{pick } Q_1(\vec{x}).\alpha_1(\vec{x}))) \neq \text{pre}(\text{pid}(\text{pick } Q_2(\vec{y}).\alpha_2(\vec{y})))$.*

Proof. Trivially true from the definition of translation $t_{\mathcal{G}}$ (see Definition 4.39). \square

We now proceed to show a property of translation $\tau_{\mathcal{G}}$ that is related to the final states of S-GKABs transition system. Roughly speaking, we show that given a final

state $s_g = \langle A_g, m_g, \delta_g \rangle$ of an S-GKAB transition system and a state $s_k = \langle A_k, m_k \rangle$ of its corresponding KAB transition system such that s_k mimics s_g (i.e., $s_g \cong s_k$), we have that there exists a state s'_k that is reachable from s_k (possibly through some intermediate states) and we have that $\text{post}(\delta_g)$ is in the ABox that is contained in s'_k (i.e., $\text{post}(\delta_g) \in \text{abox}(s'_k)$).

Lemma 4.49. *Given an S-GKAB \mathcal{G} , and a KAB $\tau_{\mathcal{G}}(\mathcal{G})$ that is obtained from \mathcal{G} through $\tau_{\mathcal{G}}$. Consider the states $s_g = \langle A_g, m_g, \delta_g \rangle$ of $\mathcal{Y}_{\mathcal{G}}^{fs}$ and $s_k = \langle A_k, m_k \rangle$ of $\mathcal{Y}_{\tau_{\mathcal{G}}(\mathcal{G})}$. If s_g is a final state, and $s_g \cong s_k$, then there exists states $\langle A_i, m_k \rangle$ and actions α_i (for $i \in \{1, \dots, n\}$, and $n \geq 0$) such that*

- $\langle A_k, m_k \rangle \xrightarrow{\alpha_1 \sigma} \langle A_1, m_k \rangle \xrightarrow{\alpha_2 \sigma} \dots \xrightarrow{\alpha_n \sigma} \langle A_n, m_k \rangle$
(with an empty substitution σ),
- $\text{State}(\text{temp}) \in A_i$ (for $i \in \{1, \dots, n\}$),
- $\text{post}(\delta_g) \in A_n$,
- $\text{pre}(\delta_g) \notin A_n$ (if $\text{post}(\delta_g) \neq \text{pre}(\delta_g)$),
- $A_n \simeq A_g$, and
- if $\text{Noop}(c) \in A_k$ (where $c \in \Delta_0$), then $\text{Noop}(c) \in A_i$ (for $i \in \{1, \dots, n\}$).

Proof. We proof the claim by induction over the definition of a final state. The rough idea about the correctness of this claim is as follows:

- Consider $\delta_g = \varepsilon$ (note that it is the base case of the definition of final state). In this case, we have that the translation of ε by $t_{\mathcal{G}}$ produces an action that simply delete the $\text{pre}(\varepsilon)$ from A_k and then add $\text{post}(\varepsilon)$. Let A_n be the resulting ABox, then it is easy to see that $A_n \simeq A_g$.
- Since $s_g \cong s_k$, based on the idea of translation $t_{\mathcal{G}}$, we have that the action execution starting from s_k should mimic the execution of program δ_g starting from s_g . Since s_g is a final state, it makes no transition to any other states. Hence, to mimic the state s_g , generalizing from the translation of ε , s_k should make transitions to the state s'_k in which $\text{post}(\delta_g) \in \text{abox}(s'_k)$. Moreover, since s_g is a final state, by the definition of final states as well as the translation $t_{\mathcal{G}}$, we have that those actions and condition-action rules that mimic δ_g only manipulate the flags. Hence it is easy to see that $\text{abox}(s'_k) \simeq A_g$.

The detail proof is as follows: Let

- $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$, and $\mathcal{Y}_{\mathcal{G}}^{fs} = \langle \Delta, T, \Sigma_g, s_{0g}, \text{abox}_g, \Rightarrow_g \rangle$,
- $\tau_{\mathcal{G}}(\mathcal{G}) = \langle T, A'_0, \Gamma', \Pi' \rangle$, and $\mathcal{Y}_{\tau_{\mathcal{G}}(\mathcal{G})} = \langle \Delta, T, \Sigma_k, s_{0k}, \text{abox}_k, \Rightarrow_k \rangle$ where
 - $A'_0 = A_0 \cup \{\text{Flag}(\text{start})\}$, and
 - $t_{\mathcal{G}}(\text{Flag}(\text{start}), \delta, \text{Flag}(\text{end})) = \langle \text{pre}, \text{post}, \Pi', \Gamma' \rangle$.

We show the claim by induction over the definition of a final state as follows:

Base case:

$[\delta_g = \varepsilon]$. Since $\langle A_g, m_g, \varepsilon \rangle \cong \langle A_k, m_k \rangle$, then by Definition 4.40 we have $\text{pre}(\varepsilon) \in A_k$.

By the definition of $t_{\mathcal{G}}$, we have a 0-ary action $\alpha_{\varepsilon}()$ where

- $\text{pre}(\varepsilon) \mapsto \alpha_{\varepsilon}()$, and
- $\text{EFF}(\alpha_{\varepsilon}) = \{\text{true} \rightsquigarrow \mathbf{add} \{\text{post}(\varepsilon), \text{State}(\text{temp})\}, \mathbf{del} \{\text{pre}(\varepsilon)\}\}$

Hence, by observing how an action is executed and the result of an action execution is constructed, we easily obtain that there exists $\langle A_1, m_k \rangle$ such that

- $\langle A_k, m_k \rangle \xrightarrow{\alpha_{\varepsilon} \sigma} \langle A_1, m_k \rangle$ (with an empty substitution σ),
- $\text{State}(\text{temp}) \in A_1$,
- $\text{post}(\varepsilon) \in A_1$,

- $pre(\varepsilon) \notin A_1$ (if $pre(\varepsilon) \neq post(\varepsilon)$), and
- $A_1 \simeq A_g$.

Additionally, it is also true that if $Noop(c) \in A_k$ (for a constant $c \in \Delta_0$), then $Noop(c) \in A_1$, because, by the definition of $t_{\mathcal{G}}$, the action α_ε does not delete any concept made by concept names **Noop** and only actions that are obtained from the translation of an action invocation delete such kind of concept assertions. Therefore the claim is proven for this case.

Inductive cases:

$[\delta_g = \delta_1 | \delta_2]$. Since $\langle A_g, m_g, \delta_1 | \delta_2 \rangle$ is a final state, then by the definition of final state (see Definition 4.14) we have either

- (1) $\langle A_g, m_g, \delta_1 \rangle$ is a final state, or
- (2) $\langle A_g, m_g, \delta_2 \rangle$ is a final state.

For compactness of the proof, here we only show the case (1). The case (2) can be done similarly. Since $\langle A_g, m_g, \delta_1 | \delta_2 \rangle \cong \langle A_k, m \rangle$, then $pre(\delta_1 | \delta_2) \in A_k$. By the definition of $\tau_{\mathcal{G}}$, we have

- $pre(\delta_1 | \delta_2) \mapsto \gamma_{\delta_1}() \in \Pi$,
- $\gamma_{\delta_1}() : \{\mathbf{true} \rightsquigarrow \mathbf{add} \{pre(\delta_1), \mathbf{State}(temp)\}, \mathbf{del} \{pre(\delta_1 | \delta_2)\}\}$,
- $post(\delta_1 | \delta_2) = post(\delta_1)$.

By observing how an action is executed as well as how the result of an action execution is constructed, we have that there exists A_{k+1} such that

- $\langle A_k, m \rangle \xrightarrow{\gamma_{\delta_1} \sigma} \langle A_{k+1}, m \rangle$,
- $\{pre(\delta_1), \mathbf{State}(temp)\} \subseteq A_{k+1}$,

Thus it is easy to see that $\langle A_g, m_g, \delta_1 \rangle \cong \langle A_{k+1}, m_k \rangle$. Then, since $\langle A_g, m_g, \delta_1 \rangle$ is a final state, by induction hypothesis, it is easy to see that the claim is proven.

$[\delta_g = \delta_1; \delta_2]$. Since $\langle A_g, m_g, \delta_1; \delta_2 \rangle$ is a final state, then by the definition of final state (see Definition 4.14) we have that $\langle A_g, m_g, \delta_1 \rangle$ is a final state and $\langle A_g, m_g, \delta_2 \rangle$ is a final state. Since $\langle A_g, m_g, \delta_1; \delta_2 \rangle \cong \langle A_k, m_k \rangle$, then $pre(\delta_1; \delta_2) \in A_k$. By the definition of $\tau_{\mathcal{G}}$ we have that $pre(\delta_1; \delta_2) = pre(\delta_1)$, $post(\delta_1) = pre(\delta_2)$, and $post(\delta_2) = post(\delta_1; \delta_2)$. By induction hypothesis, there exists states $\langle A_i, m_k \rangle$, and actions α_i , (for $i \in \{1, \dots, l\}$, and $n \geq 0$) such that

- $\langle A_k, m_k \rangle \xrightarrow{\alpha_1 \sigma} \langle A_1, m_k \rangle \xrightarrow{\alpha_2 \sigma} \dots \xrightarrow{\alpha_l \sigma} \langle A_l, m_k \rangle$
(with an empty substitution σ),
- $\mathbf{State}(temp) \in A_i$ (for $i \in \{1, \dots, l\}$),
- $post(\delta_1) \in A_l$,
- $pre(\delta_1) \notin A_l$ (if $pre(\delta_1) \neq post(\delta_1)$),
- $A_l \simeq A_g$,
- if $Noop(c) \in A_k$ (where $c \in \Delta_0$), then $Noop(c) \in A_i$ (for $i \in \{1, \dots, l\}$).

Now, since $A_l \simeq A_g$, $m_k = m_g$, $pre(\delta_2) \in A_l$, then we have $\langle A_g, m_g, \delta_2 \rangle \cong \langle A_l, m_k \rangle$. Hence, by induction hypothesis again, there exists states $\langle A_i, m_k \rangle$, and actions α_i (for $i \in \{l+1, \dots, n\}$, and $n \geq 0$) such that

- $\langle A_l, m_k \rangle \xrightarrow{\alpha_{l+1} \sigma} \langle A_{l+1}, m_k \rangle \xrightarrow{\alpha_{l+2} \sigma} \dots \xrightarrow{\alpha_n \sigma} \langle A_n, m_k \rangle$
(with an empty substitution σ),
- $\mathbf{State}(temp) \in A_i$ (for $i \in \{l+1, \dots, n\}$),

- $post(\delta_2) \in A_n$,
- $pre(\delta_2) \notin A_n$ (if $pre(\delta_2) \neq post(\delta_2)$),
- $A_n \simeq A_g$,
- if $Noop(c) \in A_l$ (where $c \in \Delta_0$),
then $Noop(c) \in A_i$ (for $i \in \{l+1, \dots, n\}$).

Therefore, it is easy to see that the claim is proven.

$[\delta_g = \text{if } \varphi \text{ then } \delta_1 \text{ else } \delta_2]$. Since $\langle A_g, m_g, \text{if } \varphi \text{ then } \delta_1 \text{ else } \delta_2 \rangle$ is a final state, then by the definition of final state (see Definition 4.14) we have either

- (1) $\langle A_g, m_g, \delta_1 \rangle$ is a final state and $ASK(\varphi, T, A) = \text{true}$, or
- (2) $\langle A_g, m_g, \delta_2 \rangle$ is a final state and $ASK(\varphi, T, A) = \text{false}$.

For compactness of the proof, here we only show the case (1). The case (2) can be done similarly. Now, since $\langle A_g, m_g, \text{if } \varphi \text{ then } \delta_1 \text{ else } \delta_2 \rangle \cong \langle A_k, m_k \rangle$, then $pre(\text{if } \varphi \text{ then } \delta_1 \text{ else } \delta_2) \in A_k$. By the definition of τ_G , we have

- $\varphi \wedge pre(\text{if } \varphi \text{ then } \delta_1 \text{ else } \delta_2) \mapsto \gamma_{if}() \in \Pi$,
- $\gamma_{if}() : \{\text{true} \rightsquigarrow \text{add } \{pre(\delta_1), \text{State}(temp)\},$
 $\quad \text{del } \{pre(\text{if } \varphi \text{ then } \delta_1 \text{ else } \delta_2)\}\},$
- $post(\text{if } \varphi \text{ then } \delta_1 \text{ else } \delta_2) = post(\delta_1)$.

By observing how an action is executed as well as how the result of an action execution is constructed, we have that there exists A_{k+1} such that

- $\langle A_k, m \rangle \xrightarrow{\gamma_{if}^\sigma} \langle A_{k+1}, m \rangle$,
- $\{pre(\delta_1), \text{State}(temp)\} \subseteq A_{k+1}$,

Thus it is easy to see that $\langle A_g, m_g, \delta_1 \rangle \cong \langle A_{k+1}, m_k \rangle$. Then, since $\langle A_g, m_g, \delta_1 \rangle$ is a final state, by induction hypothesis, it is easy to see that the claim is proven.

$[\delta_g = \text{while } \varphi \text{ do } \delta]$. Since $\langle A_g, m_g, \text{while } \varphi \text{ do } \delta \rangle$ is a final state, then by the definition of final state (see Definition 4.14) we have either

- (1) $ASK(\varphi, T, A) = \text{false}$, or
- (2) $\langle A_g, m_g, \delta \rangle$ is a final state and $ASK(\varphi, T, A) = \text{true}$.

Proof for the case (1): Now, since $\langle A_g, m_g, \text{while } \varphi \text{ do } \delta \rangle \cong \langle A_k, m_k \rangle$, then $pre(\text{while } \varphi \text{ do } \delta) \in A_k$. By the definition of τ_G , we have

- $pre(\text{while } \varphi \text{ do } \delta) \wedge (\neg\varphi \vee Noop(noop)) \mapsto \gamma_{endLoop}()$,
- $\gamma_{endLoop}() : \{\text{true} \rightsquigarrow$
 $\quad \text{add } \{post(\text{while } \varphi \text{ do } \delta), \text{State}(temp)\},$
 $\quad \text{del } \{pre(\text{while } \varphi \text{ do } \delta), Noop(noop)\}\},$

Then, by induction hypothesis, and also by observing how an action is executed as well as the result of an action execution is constructed, it is easy to see that the claim is proved.

Proof for the case (2): Now, since $\langle A_g, m_g, \text{while } \varphi \text{ do } \delta \rangle \cong \langle A_k, m_k \rangle$, then $pre(\text{while } \varphi \text{ do } \delta) \in A_k$. By the definition of τ_G , we have

- $pre(\text{while } \varphi \text{ do } \delta) \wedge \varphi \wedge \neg Noop(noop) \mapsto \gamma_{doLoop}()$,
- $pre(\text{while } \varphi \text{ do } \delta) \wedge (\neg\varphi \vee Noop(noop)) \mapsto \gamma_{endLoop}()$,
- $\gamma_{endLoop}() : \{\text{true} \rightsquigarrow$
 $\quad \text{add } \{post(\text{while } \varphi \text{ do } \delta), \text{State}(temp)\},$
 $\quad \text{del } \{pre(\text{while } \varphi \text{ do } \delta), Noop(noop)\}\},$

- $\gamma_{doLoop}() : \{\text{true} \rightsquigarrow$
 $\text{add } \{pre(\delta), \text{Noop}(\text{noop}), \text{State}(\text{temp})\},$
 $\text{del } \{pre(\text{while } \varphi \text{ do } \delta)\}\}.$

Hence, it is easy to see that we have

$$\langle A_k, m_k \rangle \xrightarrow{\gamma_{doLoop}^\sigma} \langle A'_k, m_k \rangle$$

where σ is an empty substitution, and $\{\text{State}(\text{temp}), pre(\delta), \text{Noop}(\text{noop})\} \subseteq A'_k$. Hence $\langle A_g, m_g, \delta \rangle \cong \langle A'_k, m_k \rangle$. Since $\langle A_g, m_g, \delta \rangle$ is a final state and $\langle A_g, m_g, \delta \rangle \cong \langle A'_k, m_k \rangle$, by induction hypothesis, then there exists states $\langle A_i, m_k \rangle$, and actions α_i (for $i \in \{1, \dots, n\}$, and $n \geq 0$) such that

- $\langle A'_k, m_k \rangle \xrightarrow{\alpha_1 \sigma} \langle A_1, m_k \rangle \xrightarrow{\alpha_2 \sigma} \dots \xrightarrow{\alpha_n \sigma} \langle A_n, m_k \rangle$
(with an empty substitution σ),
- $\text{State}(\text{temp}) \in A_i$ (for $i \in \{1, \dots, n\}$),
- $post(\delta) \in A_n$,
- $pre(\delta) \notin A_n$ (if $post(\delta) \neq pre(\delta)$),
- $A_n \simeq A_g$, and
- if $\text{Noop}(c) \in A'_k$ (where $c \in \Delta_0$), then $\text{Noop}(c) \in A_i$ (for $i \in \{1, \dots, n\}$).

Hence we have $\{post(\delta), \text{Noop}(\text{noop}), \text{State}(\text{temp})\} \subseteq A_n$. Now, since by the definition of $t_{\mathcal{G}}$ we have that $post(\delta) = pre(\text{while } \varphi \text{ do } \delta)$, then the action $\gamma_{endLoop}$ is executable in A_n (notice that we do not care whether $\text{ASK}(\varphi, T, A) = \text{false}$, or $\text{ASK}(\varphi, T, A) = \text{true}$ because $\text{Noop}(\text{noop}) \in A_n$). Hence we have

$$\langle A_n, m_k \rangle \xrightarrow{\gamma_{endLoop}^\sigma} \langle A'_n, m_k \rangle$$

with $\{\text{State}(\text{temp}), post(\text{while } \varphi \text{ do } \delta)\} \subseteq A'_n$, and $\text{Noop}(\text{noop}) \notin A'_n$ (which is fine since $\text{Noop}(\text{noop}) \notin A_k$). Thus we have that the claim is proven. Intuitively, the idea for the proof of this case is that since $\langle A_g, m_g, \delta \rangle$ is a final state, there is no action executed and no one removes the flag made by concept name **Noop**. In that situation, for the second iteration, no matter whether φ (the guard of the loop) holds or not, we can exit the loop and additionally keeping all assertions in the ABox (except the special markers) stay the same. Essentially it reflects the situation that in the corresponding S-GKAB, there is no transition (since $\langle A_g, m_g, \delta \rangle$ is a final state).

□

4.4.3 Reducing the Verification of S-GKABs into KABs

In this section we show that the transition system of an S-GKAB is J-bisimilar to the transition system of its corresponding KAB that is obtained via translation $\tau_{\mathcal{G}}$. Then, taking the advantage of J-Bisimulation property, we essentially show that we can reduce the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ formulas over S-GKABs into a verification over KABs.

As a start, below we show that given a state s_1 of an S-GKAB transition system, and a state s_2 of its corresponding KAB transition system such that s_2 mimics s_1 , we have that if s_1 reaches s'_1 in one step, then it implies that there exists s'_2 reachable from s_2 (possibly through some intermediate states s_1^t, \dots, s_n^t that contain $\text{State}(\text{temp})$) and s'_2 mimics s'_1 .

Lemma 4.50. *Let \mathcal{G} be an S-GKAB with transition system $\Upsilon_{\mathcal{G}}^{fs}$, $\tau_{\mathcal{G}}(\mathcal{G})$ be a KAB (obtained from \mathcal{G} through $\tau_{\mathcal{G}}$) with transition system $\Upsilon_{\tau_{\mathcal{G}}(\mathcal{G})}$. Consider two states $\langle A_g, m_g, \delta_g \rangle$ of $\Upsilon_{\mathcal{G}}^{fs}$, and $\langle A_k, m_k \rangle$ of $\Upsilon_{\tau_{\mathcal{G}}(\mathcal{G})}$ such that $\langle A_g, m_g, \delta_g \rangle \cong \langle A_k, m_k \rangle$. For every state $\langle A'_g, m'_g, \delta'_g \rangle$ such that*

$$\langle A_g, m_g, \delta_g \rangle \xrightarrow{\alpha\sigma, fs} \langle A'_g, m'_g, \delta'_g \rangle$$

(for a certain action α , and a legal parameter assignment σ), there exist states $\langle A'_k, m'_k \rangle$, $\langle A_i^t, m_k \rangle$ (for $i \in \{1, \dots, n\}$, where $n \geq 0$), and actions α' , α_i (for $i \in \{1, \dots, n\}$, where $n \geq 0$) such that

- $\langle A_k, m_k \rangle \xrightarrow{\alpha_1\sigma_e} \langle A_1^t, m_k \rangle \xrightarrow{\alpha_2\sigma_e} \dots \xrightarrow{\alpha_n\sigma_e} \langle A_n^t, m_k \rangle \xrightarrow{\alpha'\sigma} \langle A'_k, m'_k \rangle$ where
 - σ_e is an empty substitution,
 - α' is obtained from α through $t_{\mathcal{G}}$,
 - $\text{State}(\text{temp}) \in A_i^t$ (for $i \in \{1, \dots, n\}$), $\text{State}(\text{temp}) \notin A'_k$,
- $\langle A'_g, m'_g, \delta'_g \rangle \cong \langle A'_k, m'_k \rangle$.

Proof. Let

- $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$, and $\Upsilon_{\mathcal{G}}^{fs} = \langle \Delta, T, \Sigma_g, s_{0g}, \text{abox}_g, \Rightarrow_g \rangle$,
- $\tau_{\mathcal{G}}(\mathcal{G}) = \langle T, A'_0, \Gamma', \Pi' \rangle$ and $\Upsilon_{\tau_{\mathcal{G}}(\mathcal{G})} = \langle \Delta, T, \Sigma_k, s_{0k}, \text{abox}_k, \Rightarrow_k \rangle$ where
 1. $A'_0 = A_0 \cup \{\text{Flag}(\text{start})\}$, and
 2. $t_{\mathcal{G}}(\text{Flag}(\text{start}), \delta, \text{Flag}(\text{end})) = \langle \text{pre}, \text{post}, \Pi', \Gamma' \rangle$.

We prove by induction over the structure of δ .

Base case:

$[\delta_g = \varepsilon]$. Since $\langle A_g, m_g, \varepsilon \rangle$ is a final state, then there does not exist $\langle A'_g, m'_g, \delta'_g \rangle$ such that

$$\langle A_g, m_g, \delta_g \rangle \xrightarrow{\alpha\sigma, fs} \langle A'_g, m'_g, \delta'_g \rangle,$$

Hence, we do not need to show anything.

$[\delta_g = \text{pick } Q(\vec{p}).\alpha(\vec{p})]$. For compactness of the presentation, let $a = \text{pick } Q(\vec{p}).\alpha(\vec{p})$. Since $\langle A_g, m_g, a \rangle \cong \langle A_k, m_k \rangle$, then $\text{pre}(a) \in A_k$, by the definition of $\tau_{\mathcal{G}}$, we have:

- $Q(\vec{p}) \wedge \text{pre}(a) \mapsto \alpha'(\vec{p}) \in \Pi'$,
- $\alpha' \in \Gamma'$.

Since

$$\langle A_g, m_g, a \rangle \xrightarrow{\alpha\sigma, fs} \langle A'_g, m'_g, \varepsilon \rangle,$$

then $\sigma \in \text{ASK}(Q, T, A_g)$. Since $A_k \simeq A_g$, and Q does not use any special marker concept names, by Lemma 4.29 and Definition 4.4 we have $\text{ASK}(Q, T, A_g) = \text{CERT}(Q, T, A_k)$ and hence $\sigma \in \text{CERT}(Q, T, A_k)$. Now, since $\text{pre}(a) \in A_k$, then α' is executable in A_k with legal parameter assignment σ . Additionally, considering

$$\begin{aligned} \text{EFF}(\alpha') &= \text{EFF}(\alpha) \cup \{\text{true} \rightsquigarrow \mathbf{add} \{ \text{post}(a) \}, \mathbf{del} \{ \text{pre}(a), \text{State}(\text{temp}) \} \} \\ &\quad \cup \{ \text{Noop}(x) \rightsquigarrow \mathbf{del} \text{ Noop}(x) \}, \end{aligned}$$

by Definitions 3.7 and 4.7, we have $\text{ADD}(T, A_g, \alpha\sigma) = \text{ADD}(T, A_k, \alpha'\sigma) \setminus \{\text{post}(a)\}$, and hence $\text{CALLS}(\text{ADD}(T, A_g, \alpha\sigma)) = \text{CALLS}(\text{ADD}(T, A_k, \alpha'\sigma))$. Thus we have $\theta \in \text{CALLS}(\text{ADD}(T, A_k, \alpha'\sigma))$. Now, since $m'_g = \theta \cup m_g$, $m_k = m_g$ and $\theta \in \text{CALLS}(\text{ADD}(T, A_k, \alpha'\sigma))$, we can construct $m'_k = \theta \cup m_k$. Therefore it is easy to see that there exists $\langle A'_k, m'_k \rangle$, such that

$$\langle A_k, m_k \rangle \xrightarrow{\alpha'\sigma} \langle A'_k, m'_k \rangle$$

(with service call substitution θ) and $A'_g \simeq A'_k$ (by considering how A'_k is constructed), $m'_g = m'_k$. By the definition of $t_{\mathcal{G}}$ (in the translation of an action invocation) we also have $\text{pre}(\varepsilon) \in A'_k$. Thus the claim is proven.

Inductive case:

$[\delta_g = \delta_1 | \delta_2]$. Since

$$\langle A_g, m_g, \delta_1 | \delta_2 \rangle \xrightarrow{\alpha\sigma, f_S} \langle A'_g, m'_g, \delta' \rangle,$$

then, there are two cases, that is either

- (1) $\langle A_g, m_g, \delta_1 \rangle \xrightarrow{\alpha\sigma, f_S} \langle A'_g, m'_g, \delta' \rangle$, or
- (2) $\langle A_g, m_g, \delta_2 \rangle \xrightarrow{\alpha\sigma, f_S} \langle A'_g, m'_g, \delta' \rangle$.

Here we only give the derivation for the first case, the second case is similar. Since $\langle A_g, m_g, \delta_1 | \delta_2 \rangle \cong \langle A_k, m_k \rangle$, then $A_g \simeq A_k$, $m_g = m_k$, and $\text{pre}(\delta_1 | \delta_2) \in A_k$. By the definition of $\tau_{\mathcal{G}}$ and Lemma 4.41, we have

- $\text{pre}(\delta_1 | \delta_2) \mapsto \gamma_{\delta_1}() \in \Pi'$
- $\gamma_{\delta_1} \in \Gamma'$, where

$$\gamma_{\delta_1}() : \{\text{true} \rightsquigarrow \mathbf{add} \{ \text{pre}(\delta_1), \text{State}(\text{temp}) \}, \mathbf{del} \{ \text{pre}(\delta_1 | \delta_2) \} \},$$

Since $\text{pre}(\delta_1 | \delta_2) \in A_k$, it is easy to see that

$$\langle A_k, m_k \rangle \xrightarrow{\gamma_{\delta_1} \sigma_t} \langle A_t, m_k \rangle$$

where σ_t is an empty substitution, $\{\text{pre}(\delta_1), \text{State}(\text{temp})\} \in A_t$, and $A_t \simeq A_k$. Since $A_t \simeq A_k$ and $A_g \simeq A_k$, it is easy to see that $A_g \simeq A_t$. Since $A_g \simeq A_t$, $m_g = m_k$, and $\text{pre}(\delta_1) \in A_t$, then we have $\langle A_g, m_g, \delta_1 \rangle \cong \langle A_t, m_k \rangle$. Therefore, since $\langle A_g, m_g, \delta_1 \rangle \xrightarrow{\alpha\sigma, f_S} \langle A'_g, m'_g, \delta' \rangle$ and $\langle A_g, m_g, \delta_1 \rangle \cong \langle A_t, m_k \rangle$, by induction hypothesis, it is easy to see that the claim is proven for this case.

$[\delta_g = \delta_1; \delta_2]$. There are two cases:

- (1) $\langle A_g, m_g, \delta_1; \delta_2 \rangle \xrightarrow{\alpha\sigma, f_S} \langle A'_g, m'_g, \delta'_1; \delta_2 \rangle$,
- (2) $\langle A_g, m_g, \delta_1; \delta_2 \rangle \xrightarrow{\alpha\sigma, f_S} \langle A'_g, m'_g, \delta'_2 \rangle$,

Case (1). Since $\langle A_g, m_g, \delta_1; \delta_2 \rangle \xrightarrow{\alpha\sigma, f_S} \langle A'_g, m'_g, \delta'_1; \delta_2 \rangle$, then we have

$$\langle A_g, m_g, \delta_1 \rangle \xrightarrow{\alpha\sigma, f_S} \langle A'_g, m'_g, \delta'_1 \rangle,$$

Since $\langle A_g, m_g, \delta_1; \delta_2 \rangle \cong \langle A_k, m_k \rangle$, then $A_g \simeq A_k$, $m_g = m_k$, and $\text{pre}(\delta_1; \delta_2) \in A_k$. By the definition of $\tau_{\mathcal{G}}$ and Lemma 4.41, it is easy to see that $\text{pre}(\delta_1; \delta_2) = \text{pre}(\delta_1)$, and hence because $\text{pre}(\delta_1; \delta_2) \in A_k$, we have $\text{pre}(\delta_1) \in A_k$. Now, since

$A_g \simeq A_k$, $m_g = m_k$, $pre(\delta_1) \in A_k$, then we have $\langle A_g, m_g, \delta_1 \rangle \cong \langle A_k, m_k \rangle$. Thus, since we also have $\langle A_g, m_g, \delta_1 \rangle \xrightarrow{\alpha\sigma, fs} \langle A'_g, m'_g, \delta'_1 \rangle$, by using induction hypothesis we have that the claim is proven.

Case (2). Since $\langle A_g, m_g, \delta_1; \delta_2 \rangle \xrightarrow{\alpha\sigma, fs} \langle A'_g, m'_g, \delta'_2 \rangle$, then we have $\langle A_g, m_g, \delta_1 \rangle$ is a final state, and

$$\langle A_g, m_g, \delta_2 \rangle \xrightarrow{\alpha\sigma, fs} \langle A'_g, m'_g, \delta'_2 \rangle,$$

Since $\langle A_g, m_g, \delta_1 \rangle$ is a final state and $\langle A_g, m_g, \delta_g \rangle \cong \langle A_k, m_k \rangle$, by Lemma 4.49, there exist states $\langle A_i, m_k \rangle$ and actions α_i (for $i \in \{1, \dots, n\}$, and $n \geq 0$) such that

- $\langle A_k, m_k \rangle \xrightarrow{\alpha_1\sigma} \langle A_1, m_k \rangle \xrightarrow{\alpha_2\sigma} \dots \xrightarrow{\alpha_n\sigma} \langle A_n, m_k \rangle$
(with empty an empty substitution σ),
- $\text{State}(\text{temp}) \in A_i$ (for $i \in \{1, \dots, n\}$),
- $post(\delta_1) \in A_n$, $pre(\delta_1) \notin A_n$, and
- $A_n \simeq A_g$.

Since $\langle A_g, m_g, \delta_1; \delta_2 \rangle \cong \langle A_k, m_k \rangle$, then $A_g \simeq A_k$, $m_g = m_k$, and $pre(\delta_1; \delta_2) \in A_k$. By the definition of τ_G and Lemma 4.41, it is easy to see that

- $pre(\delta_1; \delta_2) = pre(\delta_1)$,
- $post(\delta_1) = pre(\delta_2)$,
- $post(\delta_1; \delta_2) = post(\delta_2)$,

Hence, because $post(\delta_1) \in A_n$, and $post(\delta_1) = pre(\delta_2)$, we have $pre(\delta_2) \in A_n$. Now, since $A_g \simeq A_n$, $m_g = m_k$, $pre(\delta_2) \in A_n$, then we have $\langle A_g, m_g, \delta_2 \rangle \cong \langle A_n, m_k \rangle$. Thus, since we also have $\langle A_g, m_g, \delta_2 \rangle \xrightarrow{\alpha\sigma, fs} \langle A'_g, m'_g, \delta'_2 \rangle$, by using induction hypothesis we have that the claim is proven.

$[\delta_g = \text{if } \varphi \text{ then } \delta_1 \text{ else } \delta_2]$. There are two cases:

- (1) $\langle A_g, m_g, \text{if } \varphi \text{ then } \delta_1 \text{ else } \delta_2 \rangle \xrightarrow{\alpha\sigma, fs} \langle A'_g, m'_g, \delta'_1 \rangle$,
- (2) $\langle A_g, m_g, \text{if } \varphi \text{ then } \delta_1 \text{ else } \delta_2 \rangle \xrightarrow{\alpha\sigma, fs} \langle A'_g, m'_g, \delta'_2 \rangle$.

Here we only consider the first case. The second case is similar.

Case (1). Since $\langle A_g, m_g, \text{if } \varphi \text{ then } \delta_1 \text{ else } \delta_2 \rangle \xrightarrow{\alpha\sigma, fs} \langle A'_g, m'_g, \delta'_1 \rangle$, then we have

$$\langle A_g, m_g, \delta_1 \rangle \xrightarrow{\alpha\sigma, fs} \langle A'_g, m'_g, \delta'_1 \rangle.$$

with $\text{ASK}(\varphi, T, A_g) = \text{true}$. Since $\langle A_g, m_g, \text{if } \varphi \text{ then } \delta_1 \text{ else } \delta_2 \rangle \cong \langle A_k, m_k \rangle$, then $A_g \simeq A_k$, $m_g = m_k$, and $pre(\text{if } \varphi \text{ then } \delta_1 \text{ else } \delta_2) \in A_k$. By the definition of τ_G and Lemma 4.41, we have

- $pre(\text{if } \varphi \text{ then } \delta_1 \text{ else } \delta_2) \wedge \varphi \mapsto \gamma_{if}() \in \Pi'$
- $\gamma_{if} \in \Gamma'$, where

$$\gamma_{if}() : \{\text{true} \rightsquigarrow \text{add } \{pre(\delta_1), \text{State}(\text{temp})\},$$

$$\text{del } \{pre(\text{if } \varphi \text{ then } \delta_1 \text{ else } \delta_2)\}\},$$

Since $A_k \simeq A_g$, $\text{ASK}(\varphi, T, A_g) = \text{true}$, and φ does not use any special marker concept names, by Lemma 4.29 and Definition 4.4 we have $\text{CERT}(\varphi, T, A_k) =$

true. Now, since $pre(\text{if } \varphi \text{ then } \delta_1 \text{ else } \delta_2) \in A_k$, and $CERT(\varphi, T, A_k) = \text{true}$, it is easy to see that

$$\langle A_k, m_k \rangle \xrightarrow{\gamma_{if}\sigma_t} \langle A_t, m_k \rangle$$

where σ_t is an empty substitution, $\{pre(\delta_1), \text{State}(temp)\} \in A_t$, and $A_t \simeq A_k$. Since $A_t \simeq A_k$ and $A_g \simeq A_k$, it is easy to see that $A_g \simeq A_t$. Since $A_g \simeq A_t$, $m_g = m_k$, and $pre(\delta_1) \in A_t$, then we have $\langle A_g, m_g, \delta_1 \rangle \cong \langle A_t, m_k \rangle$. Thus, since $\langle A_g, m_g, \delta_1 \rangle \xrightarrow{\alpha\sigma, fs} \langle A'_g, m'_g, \delta'_1 \rangle$ and $\langle A_g, m_g, \delta_1 \rangle \cong \langle A_t, m_k \rangle$, by induction hypothesis, it is easy to see that the claim is proven for this case.

$[\delta_g = \text{while } \varphi \text{ do } \delta]$. Since

$$\langle A_g, m_g, \text{while } \varphi \text{ do } \delta \rangle \xrightarrow{\alpha\sigma, fs} \langle A'_g, m'_g, \delta'; \text{while } \varphi \text{ do } \delta \rangle,$$

then we have $ASK(\varphi, T, A) = \text{true}$ and

$$\langle A_g, m_g, \delta \rangle \xrightarrow{\alpha\sigma, fs} \langle A'_g, m'_g, \delta' \rangle.$$

Since $\langle A_g, m_g, \text{while } \varphi \text{ do } \delta \rangle \cong \langle A_k, m_k \rangle$, then $A_g \simeq A_k$, $m_g = m_k$, and $pre(\text{while } \varphi \text{ do } \delta) \in A_k$. By the definition of τ_G and Lemma 4.41, we have

- $(pre(\text{while } \varphi \text{ do } \delta) \vee post(\delta)) \wedge \varphi \wedge \neg \text{Noop}(noop) \mapsto \gamma_{doLoop}() \in \Pi'$,
- $\gamma_{doLoop}() : \{\text{true} \rightsquigarrow \text{add } \{pre(\delta), \text{Noop}(noop), \text{State}(temp)\},$
 $\quad \text{del } \{pre(\text{while } \varphi \text{ do } \delta), post(\delta)\}\},$

Since $A_k \simeq A_g$, $ASK(\varphi, T, A_g) = \text{true}$, and φ does not use any special marker concept names, by Lemma 4.29 and Definition 4.4 we have $CERT(\varphi, T, A_k) = \text{true}$. Additionally, it is easy to see from the definition of t_G that $\text{Noop}(noop) \notin A_k$. Now, since $pre(\text{while } \varphi \text{ do } \delta) \in A_k$, $CERT(\varphi, T, A_k) = \text{true}$, and $\text{Noop}(noop) \notin A_k$, it is easy to see that

$$\langle A_k, m_k \rangle \xrightarrow{\gamma_{doLoop}\sigma_t} \langle A_t, m_k \rangle$$

where σ_t is an empty substitution, $\{pre(\delta), \text{State}(temp)\} \in A_t$, and $A_t \simeq A_k$. Since $A_t \simeq A_k$ and $A_g \simeq A_k$, it is easy to see that $A_g \simeq A_t$. Since $A_g \simeq A_t$, $m_g = m_k$, and $pre(\delta) \in A_t$, then we have $\langle A_g, m_g, \delta \rangle \cong \langle A_t, m_k \rangle$. Thus, since $\langle A_g, m_g, \delta \rangle \xrightarrow{\alpha\sigma, fs} \langle A'_g, m'_g, \delta' \rangle$ and $\langle A_g, m_g, \delta \rangle \cong \langle A_t, m_k \rangle$, by induction hypothesis, there exist states $\langle A'_k, m'_k \rangle$, $\langle A_i^t, m_k \rangle$ (for $i \in \{1, \dots, n\}$, where $n \geq 0$), and actions α' , α_i (for $i \in \{1, \dots, n\}$, where $n \geq 0$) such that

- $\langle A_t, m_k \rangle \xrightarrow{\alpha_1\sigma_e} \langle A_1^t, m_k \rangle \xrightarrow{\alpha_2\sigma_e} \dots \xrightarrow{\alpha_n\sigma_e} \langle A_n^t, m_k \rangle \xrightarrow{\alpha'\sigma} \langle A'_k, m'_k \rangle$ where
 - σ_e is an empty substitution,
 - α' is obtained from α through t_G ,
 - $\text{State}(temp) \in A_i^t$ (for $i \in \{1, \dots, n\}$), $\text{State}(temp) \notin A'_k$,
- $\langle A'_g, m'_g, \delta' \rangle \cong \langle A'_k, m'_k \rangle$.

The proof for this case is then completed by also observing that by the definition of program execution relation (see Definition 4.15), we have that we repeat the loop at the end of the execution of program δ , and this situation is captured in the definition of t_G by having that $post(\delta) = pre(\text{while } \varphi \text{ do } \delta)$.

□

We now proceed to show another crucial lemma for showing the bisimulation between S-GKAB transition system and the transition system of its corresponding KAB that is obtained via t_G . Basically, we show that given a state s_1 of an S-GKAB transition system, and a state s_2 of its corresponding KAB transition system such that s_2 mimics s_1 , we have that if s_2 reaches s'_2 (possibly through some intermediate states s_1^t, \dots, s_n^t that contains $\text{State}(\text{temp})$), then s_1 reaches s'_1 in one step and s'_1 is mimicked by s'_2 .

Lemma 4.51. *Let $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ be an S-GKAB with transition system $\Upsilon_{\mathcal{G}}^{fs}$, $\tau_{\mathcal{G}}(\mathcal{G}) = \langle T, A'_0, \Gamma', \Pi' \rangle$ be a KAB (obtained from \mathcal{G} through $\tau_{\mathcal{G}}$) with transition system $\Upsilon_{\tau_{\mathcal{G}}(\mathcal{G})}$, where*

1. $A'_0 = A_0 \cup \{\text{Flag}(\text{start})\}$, and
2. $t_G(\text{Flag}(\text{start}), \delta, \text{Flag}(\text{end})) = \langle \text{pre}, \text{post}, \Pi', \Gamma' \rangle$.

Consider two states $\langle A_g, m_g, \delta_g \rangle$ of $\Upsilon_{\mathcal{G}}^{fs}$, and $\langle A_k, m_k \rangle$ of $\Upsilon_{\tau_{\mathcal{G}}(\mathcal{G})}$ such that $\langle A_g, m_g, \delta_g \rangle \cong \langle A_k, m_k \rangle$. For every state $\langle A'_k, m'_k \rangle$ such that

- *there exist $\langle A_i^t, m_k \rangle$ (for $i \in \{1, \dots, n\}$, $n \geq 0$), and*
- *either we have*

$$\langle A_k, m_k \rangle \xrightarrow{\alpha_1 \sigma_e} \langle A_1^t, m_k \rangle \xrightarrow{\alpha_2 \sigma_e} \dots \xrightarrow{\alpha_n \sigma_e} \langle A_n^t, m_k \rangle \xrightarrow{\alpha' \sigma} \langle A'_k, m'_k \rangle$$

if $n > 0$ or we have

$$\langle A_k, m_k \rangle \xrightarrow{\alpha' \sigma} \langle A'_k, m'_k \rangle$$

if $n = 0$ where

- σ_e is an empty substitution,
- $\alpha_i \in \Gamma_{\varepsilon}^+$ (for $i \in \{1, \dots, n\}$),
- $\alpha' \in \Gamma_{\varepsilon}^-$,
- $\text{State}(\text{temp}) \in A_i^t$ (for $i \in \{1, \dots, n\}$), $\text{State}(\text{temp}) \notin A'_k$,

then there exists a state $\langle A'_g, m'_g, \delta'_g \rangle$ such that

- $\langle A_g, m_g, \delta_g \rangle \xrightarrow{\alpha \sigma, fs} \langle A'_g, m'_g, \delta'_g \rangle$,
- α' is obtained from the translation of a certain action invocation **pick** $Q(\vec{p}).\alpha(\vec{p})$ via t_G ,
- $\langle A'_g, m'_g, \delta'_g \rangle \cong \langle A'_k, m'_k \rangle$.

Proof. Let

- $\Upsilon_{\mathcal{G}}^{fs} = \langle \Delta, T, \Sigma_g, s_{0g}, \text{abox}_g, \Rightarrow_g \rangle$, and
- $\Upsilon_{\tau_{\mathcal{G}}(\mathcal{G})} = \langle \Delta, T, \Sigma_k, s_{0k}, \text{abox}_k, \Rightarrow_k \rangle$.

We prove by induction over the structure of δ .

Base case:

$[\delta_g = \varepsilon]$. Since $\delta_g = \varepsilon$, by the definition of t_G , there must not exist $\langle A'_k, m'_k \rangle, \langle A_i^t, m_k \rangle$ (for $i \in \{1, \dots, n\}$, and $n \geq 0$) such that either

$$\langle A_k, m_k \rangle \xrightarrow{\alpha_1 \sigma_e} \langle A_1^t, m_k \rangle \xrightarrow{\alpha_2 \sigma_e} \dots \xrightarrow{\alpha_n \sigma_e} \langle A_n^t, m_k \rangle \xrightarrow{\alpha' \sigma} \langle A'_k, m'_k \rangle$$

if $n > 0$ or

$$\langle A_k, m_k \rangle \xrightarrow{\alpha' \sigma} \langle A'_k, m'_k \rangle$$

if $n = 0$ where

- σ_e is an empty substitution,
- $\alpha_i \in \Gamma_\varepsilon^+$ (for $i \in \{1, \dots, n\}$),
- $\alpha' \in \Gamma_\varepsilon^-$,
- $\text{State}(temp) \in A_i^t$ (for $i \in \{1, \dots, n\}$), $\text{State}(temp) \notin A'_k$,

Hence, the base case is trivially true. The reason is that the translation $t_{\mathcal{G}}$ translates empty programs into an action that only adds $\text{State}(temp)$ and changes the flag. In fact, $\langle A_g, m_g, \delta_g \rangle$ is a final state, then there does not exist $\langle A'_g, m'_g, \delta'_g \rangle$ such that

$$\langle A_g, m_g, \delta_g \rangle \xrightarrow{\alpha\sigma, f_S} \langle A'_g, m'_g, \delta'_g \rangle,$$

$[\delta_g = \mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p})]$. Assume that there exist states $\langle A'_k, m'_k \rangle$, $\langle A_i^t, m_k \rangle$ (for $i \in \{1, \dots, n\}$, and $n \geq 0$) such that either

$$\langle A_k, m_k \rangle \xrightarrow{\alpha_1\sigma_e} \langle A_1^t, m_k \rangle \xrightarrow{\alpha_2\sigma_e} \dots \xrightarrow{\alpha_n\sigma_e} \langle A_n^t, m_k \rangle \xrightarrow{\alpha'\sigma} \langle A'_k, m'_k \rangle$$

if $n > 0$ or

$$\langle A_k, m_k \rangle \xrightarrow{\alpha'\sigma} \langle A'_k, m'_k \rangle$$

if $n = 0$ where

- σ_e is an empty substitution,
- $\alpha_i \in \Gamma_\varepsilon^+$ (for $i \in \{1, \dots, n\}$),
- $\alpha' \in \Gamma_\varepsilon^-$,
- $\text{State}(temp) \in A_i^t$ (for $i \in \{1, \dots, n\}$), $\text{State}(temp) \notin A'_k$,

Additionally, w.l.o.g., let θ be the corresponding substitution that evaluates service calls in the transition

$$\langle A_n^t, m_k \rangle \xrightarrow{\alpha'\sigma} \langle A'_k, m'_k \rangle$$

(Note that we consider $A_n^t = A_k$ if $n = 0$). Now, since $\langle A_g, m_g, \mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p}) \rangle \cong \langle A_k, m_k \rangle$, then $pre(\mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p})) \in A_k$. Moreover, since we also have $\delta_g = \mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p})$, by the definition of $\tau_{\mathcal{G}}$, Lemmas 4.47 and 4.48, we have that α' must be obtained from $\mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p})$ and hence we have that $Q(\vec{p}) \wedge pre(\mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p})) \mapsto \alpha'(\vec{p}) \in \Pi'$.

Now, by our assumption above and by Lemma 4.46, we have that $A_k \simeq A_1^t$, $A_i^t \simeq A_{i+1}^t$ (for $i \in \{1, \dots, n-1\}$) and hence we have $A_g \simeq A_n^t$. Then, since $A_g \simeq A_n^t$, and Q does not use any special marker concept names, by Lemma 4.29 and Definition 4.4 we have $\text{ASK}(Q, T, A_g) = \text{CERT}(Q, T, A_n^t)$ and hence $\sigma \in \text{ASK}(Q, T, A_g)$. Additionally, considering

$$\begin{aligned} \text{EFF}(\alpha') &= \text{EFF}(\alpha) \cup \{\text{true} \rightsquigarrow \mathbf{add} \ \{post(\mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p}))\}, \\ &\quad \mathbf{del} \ \{pre(\mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p})), \text{State}(temp)\}\} \\ &\quad \cup \{\text{Noop}(x) \rightsquigarrow \mathbf{del} \ \text{Noop}(x)\}, \end{aligned}$$

by Definitions 3.7 and 4.7, we have

$$\text{ADD}(T, A_g, \alpha\sigma) = \text{ADD}(T, A_n^t, \alpha'\sigma) \setminus post(\mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p})),$$

and hence $\text{CALLS}(\text{ADD}(T, A_g, \alpha\sigma)) = \text{CALLS}(\text{ADD}(T, A_n^t, \alpha'\sigma))$ and $\theta \in \text{CALLS}(\text{ADD}(T, A_g, \alpha\sigma))$. Since $m'_k = \theta \cup m_k$, $m_k = m_g$ and $\theta \in$

$\text{CALLS}(\text{ADD}(T, A_g, \alpha\sigma))$, we can construct $m'_g = \theta \cup m_g$. Thus, it is easy to see that there exists $\langle A'_g, m'_g, \varepsilon \rangle$ such that

$$\langle A_g, m_g, \mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p}) \rangle \xrightarrow{\alpha\sigma, f_S} \langle A'_g, m'_g, \varepsilon \rangle,$$

(with service call substitution θ) and $A'_g \simeq A'_k$ (by considering how A'_k is constructed), $m'_g = m'_k$. Moreover, by the definition of t_G (in the translation of an action invocation) we also have $\text{pre}(\varepsilon) \in A'_k$ (because $\text{post}(\mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p})) = \text{pre}(\varepsilon)$). Thus the claim is proven.

Inductive case:

$[\delta_g = \delta_1|\delta_2]$. Assume that there exist states $\langle A'_k, m'_k \rangle, \langle A_i^t, m_k \rangle$ (for $i \in \{1, \dots, n\}$, and $n \geq 0$) such that

$$\langle A_k, m_k \rangle \xrightarrow{\alpha_1\sigma_e} \langle A_1^t, m_k \rangle \xrightarrow{\alpha_2\sigma_e} \dots \xrightarrow{\alpha_n\sigma_e} \langle A_n^t, m_k \rangle \xrightarrow{\alpha'\sigma} \langle A'_k, m'_k \rangle$$

where

- σ_e is an empty substitution,
- $\alpha_i \in \Gamma_\varepsilon^+$ (for $i \in \{1, \dots, n\}$),
- $\alpha' \in \Gamma_\varepsilon^-$,
- $\text{State}(\text{temp}) \in A_i^t$ (for $i \in \{1, \dots, n\}$), $\text{State}(\text{temp}) \notin A'_k$,

Now, by the definition of t_G on the translation of a program of the form $\delta_1|\delta_2$ and $\mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p})$, there exists $j \in \{1, \dots, n\}$ such that $\text{pre}(\delta_1|\delta_2) \in A_{j-1}^t$, and either

- a) $\alpha_j = \gamma_{\delta_1}$, and $\text{pre}(\delta_1) \in A_j^t$, or
- b) $\alpha_j = \gamma_{\delta_2}$, and $\text{pre}(\delta_2) \in A_j^t$.

where γ_{δ_1} and γ_{δ_2} are the actions obtained from the translation of $\delta_1|\delta_2$ by t_G , and note that we consider $A_k = A_{j-1}^t$ if $j = 1$. Now, by our assumption above and by Lemma 4.46, we have that $A_k \simeq A_j^t$, and hence it is easy to see that we also have $A_g \simeq A_j^t$. Thus, essentially we have

$$\langle A_{j-1}^t, m_k \rangle \xrightarrow{\alpha_j\sigma_e} \langle A_j^t, m_k \rangle \xrightarrow{\alpha_{j+1}\sigma_e} \dots \xrightarrow{\alpha_n\sigma_e} \langle A_n^t, m_k \rangle \xrightarrow{\alpha'\sigma} \langle A'_k, m'_k \rangle,$$

and

- a) if $\alpha_j = \gamma_{\delta_1}$, then $\langle A_g, m_g, \delta_1 \rangle \cong \langle A_j^t, m_k \rangle$ (because $A_g \simeq A_j^t$, $m_g = m_k$, $\text{pre}(\delta_1) \in A_j^t$), otherwise
- b) if $\alpha_j = \gamma_{\delta_2}$, then $\langle A_g, m_g, \delta_2 \rangle \cong \langle A_j^t, m_k \rangle$ (because $A_g \simeq A_j^t$, $m_g = m_k$, $\text{pre}(\delta_2) \in A_j^t$).

Therefore by induction hypothesis, it is easy to see that the claim is proven.

$[\delta_g = \delta_1;\delta_2]$. Assume that there exist states $\langle A'_k, m'_k \rangle, \langle A_i^t, m_k \rangle$ (for $i \in \{1, \dots, n\}$, and $n \geq 0$) such that

$$\langle A_k, m_k \rangle \xrightarrow{\alpha_1\sigma_e} \langle A_1^t, m_k \rangle \xrightarrow{\alpha_2\sigma_e} \dots \xrightarrow{\alpha_n\sigma_e} \langle A_n^t, m_k \rangle \xrightarrow{\alpha'\sigma} \langle A'_k, m'_k \rangle$$

where

- σ_e is an empty substitution,
- $\alpha_i \in \Gamma_\varepsilon^+$ (for $i \in \{1, \dots, n\}$),
- $\alpha' \in \Gamma_\varepsilon^-$,
- $\text{State}(\text{temp}) \in A_i^t$ (for $i \in \{1, \dots, n\}$), $\text{State}(\text{temp}) \notin A'_k$,

By the definition of t_G on the translation of $\delta_1; \delta_2$ and **pick** $Q(\vec{p}).\alpha(\vec{p})$, as well as Lemma 4.49, then there are two cases:

- (a) The case when $\langle A_g, m_g, \delta_1 \rangle$ is not a final state. Either
- $\text{pre}(\delta_1) \in A_k$ and $\text{pre}(\delta_1) \notin A_j^t$ for $j \in \{1, \dots, n\}$, or
 - there exists $j \in \{1, \dots, n\}$ such that $\text{pre}(\delta_1) \in A_j^t$, and $\text{pre}(\delta_1) \notin A_l^t$ for $l \in \{j+1, \dots, n\}$.
- (b) The case when $\langle A_g, m_g, \delta_1 \rangle$ is a final state. There exists $j \in \{1, \dots, n-1\}$ and $l \in \{j+1, \dots, n\}$ such that $\text{pre}(\delta_1) \in A_j^t$, $\text{post}(\delta_1) \in A_l^t$, $\text{pre}(\delta_2) \in A_l^t$, $\text{post}(\delta_1) = \text{pre}(\delta_2)$.

Now, by our assumption above and by Lemma 4.46, we have that

- **For the case (a):** either

- $A_g \simeq A_k$ and thus we have that $\langle A_g, m_g, \delta_1 \rangle \cong \langle A_k, m_k \rangle$, or
- $A_k \simeq A_j^t$, and hence it is easy to see that we also have $A_g \simeq A_j^t$. Thus we have that $\langle A_g, m_g, \delta_1 \rangle \cong \langle A_j^t, m_k \rangle$.

- **For the case (b):** $A_k \simeq A_l^t$, and hence it is easy to see that we also have $A_g \simeq A_l^t$. Thus we have that $\langle A_g, m_g, \delta_2 \rangle \cong \langle A_l^t, m_k \rangle$.

Therefore by induction hypothesis, it is easy to see that the claim is proven.

$[\delta_g = \text{if } \varphi \text{ then } \delta_1 \text{ else } \delta_2]$. Assume that there exist states $\langle A'_k, m'_k \rangle$, $\langle A_i^t, m_k \rangle$ (for $i \in \{1, \dots, n\}$, and $n \geq 0$) such that

$$\langle A_k, m_k \rangle \xrightarrow{\alpha_1 \sigma_e} \langle A_1^t, m_k \rangle \xrightarrow{\alpha_2 \sigma_e} \dots \xrightarrow{\alpha_n \sigma_e} \langle A_n^t, m_k \rangle \xrightarrow{\alpha' \sigma} \langle A'_k, m'_k \rangle$$

where

- σ_e is an empty substitution,
- $\alpha_i \in \Gamma_\varepsilon^+$ (for $i \in \{1, \dots, n\}$),
- $\alpha' \in \Gamma_\varepsilon^-$,
- $\text{State}(\text{temp}) \in A_i^t$ (for $i \in \{1, \dots, n\}$), $\text{State}(\text{temp}) \notin A'_k$,

By the definition of t_G on the translation of a program of the form **if** φ **then** δ_1 **else** δ_2 and **pick** $Q(\vec{p}).\alpha(\vec{p})$, there exists $j \in \{1, \dots, n-1\}$ such that $\text{pre}(\text{if } \varphi \text{ then } \delta_1 \text{ else } \delta_2) \in A_{j-1}^t$ (note that we consider $A_k = A_{j-1}^t$ if $j = 1$) and either

- $\alpha_j = \gamma_{if}$, $\text{pre}(\delta_1) \in A_j^t$, and $\text{CERT}(\varphi, T, A_{j-1}^t) = \text{true}$, or
- $\alpha_j = \gamma_{else}$, $\text{pre}(\delta_2) \in A_j^t$, and $\text{CERT}(\varphi, T, A_{j-1}^t) = \text{false}$.

where γ_{if} and γ_{else} are the actions obtained from the translation of **if** φ **then** δ_1 **else** δ_2 by t_G . Now, by our assumption above and by Lemma 4.46, we have that $A_k \simeq A_j^t$, and hence it is easy to see that we also have $A_g \simeq A_j^t$. Thus, essentially we have

$$\langle A_{j-1}^t, m_k \rangle \xrightarrow{\alpha_j \sigma_e} \langle A_j^t, m_k \rangle \xrightarrow{\alpha_{j+1} \sigma_e} \dots \xrightarrow{\alpha_n \sigma_e} \langle A_n^t, m_k \rangle \xrightarrow{\alpha' \sigma} \langle A'_k, m'_k \rangle,$$

and

- a) if $\alpha_j = \gamma_{if}$, then $\langle A_g, m_g, \delta_1 \rangle \cong \langle A_j^t, m_k \rangle$ (because $A_g \simeq A_j^t$, $m_g = m_k$, $pre(\delta_1) \in A_j^t$), otherwise
- b) if $\alpha_j = \gamma_{else}$, then $\langle A_g, m_g, \delta_2 \rangle \cong \langle A_j^t, m_k \rangle$ (because $A_g \simeq A_j^t$, $m_g = m_k$, $pre(\delta_2) \in A_j^t$).

Therefore by induction hypothesis, it is easy to see that the claim is proven.

$[\delta_g = \mathbf{while} \varphi \mathbf{do} \delta_1]$. Assume that there exist states $\langle A'_k, m'_k \rangle$, $\langle A_i^t, m_k \rangle$ (for $i \in \{1, \dots, n\}$, and $n \geq 0$) such that

$$\langle A_k, m_k \rangle \xrightarrow{\alpha_1 \sigma_e} \langle A_1^t, m_k \rangle \xrightarrow{\alpha_2 \sigma_e} \dots \xrightarrow{\alpha_n \sigma_e} \langle A_n^t, m_k \rangle \xrightarrow{\alpha' \sigma} \langle A'_k, m'_k \rangle$$

where

- σ_e is an empty substitution,
- $\alpha_i \in \Gamma_\varepsilon^+$ (for $i \in \{1, \dots, n\}$),
- $\alpha' \in \Gamma_\varepsilon^-$,
- $\mathbf{State}(temp) \in A_i^t$ (for $i \in \{1, \dots, n\}$), $\mathbf{State}(temp) \notin A'_k$,

By the definition of $t_{\mathcal{G}}$ on the translation of a program of the form **while** φ **do** δ_1 and **pick** $Q(\vec{p}).\alpha(\vec{p})$, there exists $j \in \{1, \dots, n-1\}$ such that

- $\alpha_j = \gamma_{doLoop}$ (γ_{doLoop} is the action obtained during the translation of **while** φ **do** δ_1 by $t_{\mathcal{G}}$),
- $pre(\mathbf{while} \varphi \mathbf{do} \delta_1) \in A_{j-1}^t$ (where $A_{j-1}^t = A_k$ when $j = 1$), and
- $pre(\delta_1) \in A_j^t$.

Now, by our assumption above and by Lemma 4.46, we have that $A_k \simeq A_j^t$, and hence it is easy to see that $A_g \simeq A_j^t$. Thus, essentially we have

$$\langle A_{j-1}^t, m_k \rangle \xrightarrow{\alpha_j \sigma_e} \langle A_j^t, m_k \rangle \xrightarrow{\alpha_{j+1} \sigma_e} \dots \xrightarrow{\alpha_n \sigma_e} \langle A_n^t, m_k \rangle \xrightarrow{\alpha' \sigma} \langle A'_k, m'_k \rangle,$$

and $\langle A_g, m_g, \delta_1 \rangle \cong \langle A_j^t, m_k \rangle$ (because $A_g \simeq A_j^t$, $m_g = m_k$, $pre(\delta_1) \in A_j^t$). Therefore by induction hypothesis, it is easy to see that the claim is proven. \square

Now we will show that given a state s_g of an S-GKAB transition system and a state s_k of its corresponding KAB transition system such that s_g is mimicked by s_k , then we have s_g and s_k are J-bisimilar. Formally this claim is stated and shown below.

Lemma 4.52. *Let \mathcal{G} be an S-GKAB with transition system $\Upsilon_{\mathcal{G}}^{fs}$, and let $\tau_{\mathcal{G}}(\mathcal{G})$ be a KAB with transition system $\Upsilon_{\tau_{\mathcal{G}}(\mathcal{G})}$ obtained from \mathcal{G} through $\tau_{\mathcal{G}}$. Consider a state $\langle A_g, m_g, \delta_g \rangle$ of $\Upsilon_{\mathcal{G}}^{fs}$ and a state $\langle A_k, m_k \rangle$ of $\Upsilon_{\tau_{\mathcal{G}}(\mathcal{G})}$. If $\langle A_g, m_g, \delta_g \rangle \cong \langle A_k, m_k \rangle$ then $\langle A_g, m_g, \delta_g \rangle \sim_J \langle A_k, m_k \rangle$.*

Proof. Let

- $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ and $\Upsilon_{\mathcal{G}}^{fs} = \langle \Delta, T, \Sigma_g, s_{0g}, abox_g, \Rightarrow_g \rangle$,
- $\tau_{\mathcal{G}}(\mathcal{G}) = \langle T, A'_0, \Gamma', \Pi' \rangle$, and $\Upsilon_{\tau_{\mathcal{G}}(\mathcal{G})} = \langle \Delta, T, \Sigma_k, s_{0k}, abox_k, \Rightarrow_k \rangle$

We have to show:

- (1) for every state $\langle A'_g, m'_g, \delta'_g \rangle$ such that $\langle A_g, m_g, \delta_g \rangle \Rightarrow_g \langle A'_g, m'_g, \delta'_g \rangle$, there exist states $\langle A'_k, m'_k \rangle, \langle A_1^t, m_k \rangle \dots \langle A_n^t, m_k \rangle$ (for $n \geq 0$) with

$$\langle A_k, m_k \rangle \Rightarrow_k \langle A_1^t, m_k \rangle \Rightarrow_k \dots \Rightarrow_k \langle A_n^t, m_k \rangle \Rightarrow_k \langle A'_k, m'_k \rangle$$

such that:

- a) $\text{State}(\text{temp}) \notin A'_k$, $\text{State}(\text{temp}) \in A_i^t$ for $i \in \{1, \dots, n\}$, and
 - b) $\langle A'_g, m'_g, \delta'_g \rangle \cong \langle A'_k, m'_k \rangle$.
- (2) for every state $\langle A'_k, m'_k \rangle$ such that there exist states $\langle A_1^t, m_1 \rangle \dots \langle A_n^t, m_n \rangle$ (for $n \geq 0$) and

$$\langle A_k, m_k \rangle \Rightarrow_k \langle A_1^t, m_1 \rangle \Rightarrow_k \dots \Rightarrow_k \langle A_n^t, m_n \rangle \Rightarrow_k \langle A'_k, m'_k \rangle$$

where $\text{State}(\text{temp}) \notin A'_k$, and $\text{State}(\text{temp}) \in A_i^t$ for $i \in \{1, \dots, n\}$, then there exists a state $\langle A'_g, m'_g, \delta'_g \rangle$ with $\langle A_g, m_g, \delta_g \rangle \Rightarrow_g \langle A'_g, m'_g, \delta'_g \rangle$ such that $\langle A'_g, m'_g, \delta'_g \rangle \cong \langle A'_k, m'_k \rangle$.

Proof for (1): Assume $\langle A_g, m_g, \delta_g \rangle \Rightarrow \langle A'_g, m'_g, \delta'_g \rangle$, then by Definition 4.16 we have

$$\langle A_g, m_g, \delta_g \rangle \xrightarrow{\alpha\sigma, fs} \langle A'_g, m'_g, \delta'_g \rangle.$$

Additionally, by Definitions 4.12 and 4.15 we have that A'_g is T -consistent. By Lemma 4.50, there exist states $\langle A_i^t, m_k \rangle$, and actions α_i (for $i \in \{1, \dots, n\}$, where $n \geq 0$) such that

- $\langle A_k, m_k \rangle \xrightarrow{\alpha_1\sigma_e} \langle A_1^t, m_k \rangle \xrightarrow{\alpha_2\sigma_e} \dots \xrightarrow{\alpha_n\sigma_e} \langle A_n^t, m_k \rangle \xrightarrow{\alpha'\sigma} \langle A'_k, m'_k \rangle$ where
 - σ_e is an empty substitution,
 - α' is obtained from α through $t_{\mathcal{G}}$,
 - $\text{State}(\text{temp}) \in A_i^t$ (for $i \in \{1, \dots, n\}$), $\text{State}(\text{temp}) \notin A'_k$
- $\langle A'_g, m'_g, \delta'_g \rangle \cong \langle A'_k, m'_k \rangle$,

Additionally, since A'_g is T -consistent and $A'_g \simeq A'_k$ then A'_k is T -consistent. As a consequence, we have that the claim is easily proven, since by Definition 3.11, we have

$$\langle A_k, m_k \rangle \Rightarrow_k \langle A_1^t, m_k \rangle \Rightarrow_k \dots \Rightarrow_k \langle A_n^t, m_k \rangle \Rightarrow_k \langle A'_k, m'_k \rangle$$

where

- a) $\text{State}(\text{temp}) \notin A'_k$, and $\text{State}(\text{temp}) \in A_i^t$ for $i \in \{1, \dots, n\}$, and
- b) $\langle A'_g, m'_g, \delta'_g \rangle \cong \langle A'_k, m'_k \rangle$,

Proof for (2): Assume

$$\langle A_k, m_k \rangle \Rightarrow_k \langle A_1^t, m_k \rangle \Rightarrow_k \dots \Rightarrow_k \langle A_n^t, m_k \rangle \Rightarrow_k \langle A'_k, m'_k \rangle$$

where $n \geq 0$, $\text{State}(\text{temp}) \notin A'_k$, and $\text{State}(\text{temp}) \in A_i^t$ for $i \in \{1, \dots, n\}$. By Definition 3.11, we have

$$\langle A_k, m_k \rangle \xrightarrow{\alpha_1\sigma_1} \langle A_1^t, m_1 \rangle \xrightarrow{\alpha_2\sigma_2} \dots \xrightarrow{\alpha_n\sigma_n} \langle A_n^t, m_n \rangle \xrightarrow{\alpha'\sigma'} \langle A'_k, m'_k \rangle.$$

For some actions α' , α_i (for $i \in \{1, \dots, n\}$), and substitutions σ' , σ_i (for $i \in \{1, \dots, n\}$). Let Γ_{ε}^+ (resp. Γ_{ε}^-) be the set of temp adder (resp. deleter) actions of $\tau_{\mathcal{G}}(\mathcal{G})$, since $\text{State}(\text{temp}) \notin A'_k$, and $\text{State}(\text{temp}) \in A_i^t$ for $i \in \{1, \dots, n\}$, by Lemmas 4.46 and 4.47, we have that

- σ_i is an empty substitution (for $i \in \{1, \dots, n\}$).
- $\alpha_i \in \Gamma_\varepsilon^+$ (for $i \in \{1, \dots, n\}$), and
- $\alpha' \in \Gamma_\varepsilon^-$,
- there exists an action invocation **pick** $Q(\vec{p}).\alpha(\vec{p})$ that is a sub-program of δ such that α' is obtained from the translation of **pick** $Q(\vec{p}).\alpha(\vec{p})$ by $t_{\mathcal{G}}$,
- α_i (for $i \in \{1, \dots, n\}$) does not involve any service calls, and hence $m_i = m_{i+1}$ (for $i \in \{1, \dots, n-1\}$).

Therefore by Lemma 4.51, then there exists a state $\langle A'_g, m'_g, \delta'_g \rangle$ such that

- $\langle A_g, m_g, \delta_g \rangle \xrightarrow{\alpha\sigma, fs} \langle A'_g, m'_g, \delta'_g \rangle$,
- α' is obtained from the translation of a certain action invocation **pick** $Q(\vec{p}).\alpha(\vec{p})$ via $t_{\mathcal{G}}$.
- $\langle A'_g, m'_g, \delta'_g \rangle \cong \langle A'_k, m'_k \rangle$.

Since $\langle A_g, m_g, \delta_g \rangle \xrightarrow{\alpha\sigma, fs} \langle A'_g, m'_g, \delta'_g \rangle$, by Definition 4.16, we have that $\langle A_g, m_g, \delta_g \rangle \Rightarrow \langle A'_g, m'_g, \delta'_g \rangle$. Thus it is easy that the claim is proven since we also have that $\langle A'_g, m'_g, \delta'_g \rangle \cong \langle A'_k, m'_k \rangle$.

□

Having Lemma 4.52 in hand, we can easily show that given an S-GKAB, its transition system is J-bisimilar to the transition system of its corresponding KAB that is obtained via the translation $\tau_{\mathcal{G}}$.

Lemma 4.53. *Given an S-GKAB \mathcal{G} , let $\tau_{\mathcal{G}}(\mathcal{G})$ be the KAB obtained from \mathcal{G} by applying the translation $\tau_{\mathcal{G}}$, we have $\Upsilon_{\mathcal{G}}^{fs} \sim_J \Upsilon_{\tau_{\mathcal{G}}(\mathcal{G})}$*

Proof. Let

1. $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$, and $\Upsilon_{\mathcal{G}}^{fs} = \langle \Delta, T, \Sigma_g, s_{0g}, abox_g, \Rightarrow_g \rangle$,
2. $\tau_{\mathcal{G}}(\mathcal{G}) = \langle T, A'_0, \Gamma', \Pi' \rangle$, and $\Upsilon_{\tau_{\mathcal{G}}(\mathcal{G})} = \langle \Delta, T, \Sigma_k, s_{0k}, abox_k, \Rightarrow_k \rangle$

We have that $s_{0g} = \langle A_0, m_g, \delta_g \rangle$ and $s_{0k} = \langle A'_0, m_k \rangle$ where $m_g = m_k = \emptyset$. Since $A'_0 = A_0 \cup \{\text{Flag}(\text{start})\}$, and **Flag** is a special vocabulary outside the vocabulary of T , hence $A'_0 \simeq A_0$. Now, by Lemma 4.41, we have $\text{pre}(\delta) = \text{Flag}(\text{start})$ and $\text{post}(\delta) = \text{Flag}(\text{end})$. Furthermore, since $\text{Flag}(\text{start}) \in A'_0$, then we have $s_{0g} \cong s_{0k}$. Hence by Lemma 4.52, we have $s_{0g} \sim_J s_{0k}$. Therefore, we have $\Upsilon_{\mathcal{G}}^{fs} \sim_J \Upsilon_{\tau_{\mathcal{G}}(\mathcal{G})}$.

□

Having all of these machinery in hand, we are now ready to show that the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ properties over S-GKABs can be recast as verification over KAB as follows.

Theorem 4.54. *Given an S-GKAB \mathcal{G} and a closed $\mu\mathcal{L}_A^{\text{EQL}}$ property Φ in NNF, let $\tau_{\mathcal{G}}(\mathcal{G})$ be the KAB obtained from \mathcal{G} by applying the translation $\tau_{\mathcal{G}}$, we have*

$$\Upsilon_{\mathcal{G}}^{fs} \models \Phi \text{ if and only if } \Upsilon_{\tau_{\mathcal{G}}(\mathcal{G})} \models t_j(\Phi)$$

Proof. By Lemma 4.53, we have that $\Upsilon_{\mathcal{G}}^{fs} \sim_J \Upsilon_{\tau_{\mathcal{G}}(\mathcal{G})}$. Hence, by Lemma 4.33, we have that for every $\mu\mathcal{L}_A^{\text{EQL}}$ property Φ

$$\Upsilon_{\mathcal{G}}^{fs} \models \Phi \text{ if and only if } \Upsilon_{\tau_{\mathcal{G}}(\mathcal{G})} \models t_j(\Phi)$$

□

4.4.4 Verification of Run-Bounded S-GKABs

An interesting property of the translation $\tau_{\mathcal{G}}$ is that it preserves run-boundedness.

Lemma 4.55. *Let \mathcal{G} be an arbitrary S-GKAB and $\tau_{\mathcal{G}}(\mathcal{G})$ be its corresponding KAB obtained through the translation $\tau_{\mathcal{G}}$. We have that \mathcal{G} is run-bounded if and only if $\tau_{\mathcal{G}}(\mathcal{G})$ is run-bounded.*

Proof. Let $\mathcal{R}_{\mathcal{G}}^{fs}$ be the transition system of \mathcal{G} , and $\mathcal{R}_{\tau_{\mathcal{G}}(\mathcal{G})}$ be the transition system of $\tau_{\mathcal{G}}(\mathcal{G})$. The proof is then easily obtained since

- only a bounded number of new constants are introduced when emulating the Golog program with KAB condition-action rules and actions. This fact can be easily seen by observing the following:
 - Given a Golog program δ there are only finitely many sub-programs (i.e., it only yields finite number of program IDs).
 - As it can be seen from the definition of program translation $t_{\mathcal{G}}$ (cf. Definition 4.39), which recursively translates each sub-program, for each case of the sub-program translation, we only introduce finitely many fresh constants.
- by Lemma 4.53, we have that $\mathcal{R}_{\mathcal{G}}^{fs} \sim_{\mathcal{J}} \mathcal{R}_{\tau_{\mathcal{G}}(\mathcal{G})}$. Thus, basically they are “equivalent” modulo intermediate states (states containing $\text{State}(\text{temp})$), and each two bisimilar states are equivalent modulo special markers.

□

Now, we can easily acquire the following result on verification of $\mu\mathcal{L}_A^{\text{EQL}}$ properties over run-bounded S-GKABs.

Theorem 4.56 (Verification of Run-Bounded S-GKABs). *Verification of closed $\mu\mathcal{L}_A^{\text{EQL}}$ formulas over a run-bounded S-GKAB is decidable and can be reduced to finite-state model checking.*

Proof. From Theorem 4.54 and Lemma 4.55, we have that verification of closed $\mu\mathcal{L}_A^{\text{EQL}}$ formulas over run-bounded S-GKABs can be reduced to the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ formulas over run-bounded KABs. Then, by Theorem 3.30, we have that verification of $\mu\mathcal{L}_A^{\text{EQL}}$ over run-bounded KABs are decidable and can be reduced to finite-state model checking. □

4.5 Discussion

As we have seen above, here we only consider some constructs of typical Golog program (cf. [134, 90]). However, we are able to simulate some other Golog program constructs within GKABs. In this section, we discuss some of those possibilities.

Nondeterministic iteration (i.e., δ^*) is seamlessly supported by G-KABs, using the core set of constructs we considered. In fact, δ^* can be simulated as

while true do δ_1

where $\delta_1 = \delta \mid \varepsilon$. Notice that essentially the meaning of δ^* is that we execute δ zero or more times. Thus, in the while loop above, by having $\delta \mid \varepsilon$ within the body of the loop we can either choose:

1. to execute δ (and it can be done as much as we want).
2. to consider the program as completed and move to next instruction (Note that by the definition of final state, we have that

$$\langle A, m, \text{while true do } \delta_1 \rangle \in \mathbb{F}$$

for any ABox A and service call map m).

Regarding the test construct (i.e., $\varphi?$), based on the semantics of test construct in [134, 90, 89], roughly speaking, the meaning of the test construct $\varphi?$ is that when the test φ is satisfied, then the system makes a transition into a state where the data stay the same but the remaining program to be executed become ε . I.e., within our setting we can extend our program execution relation definition by adding the following:

$$\langle A, m, \varphi? \rangle \xrightarrow{\alpha\sigma, f} \langle A, m, \varepsilon \rangle, \text{ if } \text{ASK}(\varphi, T, A) = \text{true};$$

it is easy to see that such construct can be simulated by

$$\text{if } \varphi \text{ then } \delta_1 \text{ else } \delta_2$$

where:

- $\delta_1 = \text{pick true}.\alpha_{DoNothing}()$, and $\alpha_{DoNothing}$ is an action that does not change the ABox,
- $\delta_2 = \text{pick false}.\alpha_{Block}()$, and α_{Block} is any action (the idea is just to block the execution and basically it will be blocked in any case since the guard is **false**).

The idea of the program above is as follows:

- in case φ is satisfied, the system will execute the action $\alpha_{DoNothing}$ which will not change the data but the remaining program to be executed becomes ε (i.e., the system can progress further to execute the next program instruction).
- in case φ is not satisfied, the system execution will be blocked since the action α_{Block} in any case can not be executed.

Moreover, we can also simulate another semantics of test construct $\varphi?$ as in [185, 84]. According to [185, 84], the program made by a test construct is considered to be completed when the test is satisfied. Technically the state is considered to be final state (i.e., completed) when the test is satisfied. Therefore, within our setting, we can extend our definition of final state by adding the following:

$$\langle A, m, \varphi? \rangle \in \mathbb{F}, \text{ if } \text{ASK}(\varphi, T, A) = \text{true};$$

In this case, we can simulate such construct as follows:

$$\text{if } \varphi \text{ then } \varepsilon \text{ else pick false}.\alpha_{Block}()$$

where α_{Block} is any action. The idea of the program above is as follows:

- in case φ is satisfied, since the “if case” goes to ε , then according to the definition of final states, the corresponding state that has the program **if** φ **then** ε **else** δ can be considered as a final state (i.e., completed).
- in case the φ is not satisfied, the system execution will be blocked since the action α_{Block} in any case can not be executed because it is guarded with **false**.

About the pick construct, so far our GKABs only consider the pick construct that ranges over an action. However, one might easily extend it into the pick construct that ranges over a program, and GKABs with such extension can actually be simulated by our GKABs that only consider the pick construct that ranges over an action. We provide the ideas below.

Basically, we can extend our definition of Golog program (cf. Definition 4.1) by allowing the following pick construct:

$$\mathbf{pick} \ Q(\vec{x}).\delta[\vec{x}]$$

that

1. picks a tuple \vec{c} in the answer of $Q(\vec{x})$,
2. instantiates the rest of the program δ by substituting \vec{x} with \vec{c} , and
3. then executes δ .

Notice that each $x \in \vec{x}$ that occurs in a query within δ must be a free variable (recall that the idea of pick is to pick a constant to instantiate the action). We can extend our definition of the program execution relation (cf. Definition 4.15) by adding the following:

- $\langle A, m, \mathbf{pick} \ Q(\vec{x}).\delta[\vec{x}] \rangle \xrightarrow{\alpha\sigma, f} \langle A', m', \delta'[\vec{x}/\vec{c}] \rangle$,
if $\vec{c} \in \text{ASK}(Q(\vec{x}), T, A)$, and $\langle A, m, \delta[\vec{x}/\vec{c}] \rangle \xrightarrow{\alpha\sigma, f} \langle A', m', \delta'[\vec{x}/\vec{c}] \rangle$,

where $\delta[\vec{x}/\vec{c}]$ means that we substitute the variables \vec{x} in δ with \vec{c} .

We first illustrate the idea of translating GKABs with such extension into our GKABs as follows: Consider the program:

$$\delta = \mathbf{pick} \ Q(x_1, x_2).\delta_1[x_1, x_2]$$

where

$$\delta_1 = \mathbf{pick} \ Q_1(x_1, x_2, x_3).\alpha_1(x_1, x_2, x_3); \mathbf{pick} \ Q_2(x_1, x_4).\alpha_2(x_1, x_4).$$

Let $\langle a, b \rangle$ be an answer of $Q(x_1, x_2)$ (i.e., x_1 (resp. x_2) is substituted by a (resp. b)). Hence, because we need to substitute each occurrence of x_1 (resp. x_2) within δ with a (resp. b), we basically have the following:

- to instantiate the parameters of α_1 , we must only consider those answers of Q_1 in which x_1 (resp. x_2) is substituted by a (resp. b);
- similarly, to instantiate the parameters of α_2 , we must only consider those answers of Q_2 in which x_1 is substituted by a .

To simulate such situation, the idea is as follows:

1. We use some temporary ABox assertions to keep the answer of Q that is used to instantiate x_1 and x_2 in δ' . In our example, we can introduce two fresh concepts $V_{x_1}^Q$ and $V_{x_2}^Q$ to store the picked values for x_1 and x_2 .
2. We change each action invocation within δ' such that when they want to get a value for x_1 (resp. x_2), they should only consider the value in $V_{x_1}^Q$ (resp. $V_{x_2}^Q$). To do this, we can just conjunct each query in each action invocation with the query that retrieves values from $V_{x_1}^Q$ and $V_{x_2}^Q$ (i.e., $Q^x(x_1, x_2) = V_{x_1}^Q(x_1) \wedge V_{x_2}^Q(x_2)$). Essentially, we can translate δ_1 into the following:

$$\begin{aligned} \delta'_1 = & \mathbf{pick} \ Q_1(x_1, x_2, x_3) \wedge Q^x(x_1, x_2).\alpha_1(x_1, x_2, x_3); \\ & \mathbf{pick} \ Q_2(x_1, x_4) \wedge Q^x(x_1, x_2).\alpha_2(x_1, x_4). \end{aligned}$$

Note that in the second atomic action invocation above (i.e., that execute α_2), we need to abuse our definition of atomic action invocation because we pick more values than what is needed in the action parameters of α_2 (i.e., we have an atomic action invocation of the form **pick** $Q(\vec{y}).\alpha(\vec{x})$ where $\vec{x} \subseteq \vec{y}$). However, considering the form of the query Q_x , it is easy to see that we can do post processing in our translation to remove unnecessary query. I.e., we can simply translate δ'_1 into

$$\begin{aligned} \delta''_1 = & \textbf{pick } Q_1(x_1, x_2, x_3) \wedge V_{x_1}^Q(x_1) \wedge V_{x_2}^Q(x_2). \alpha_1(x_1, x_2, x_3); \\ & \textbf{pick } Q_2(x_1, x_4) \wedge V_{x_1}^Q(x_1). \alpha_2(x_1, x_4). \end{aligned}$$

3. At the end of the execution of δ'_1 we need to delete those temporary ABox assertions that was used to store the picked values (i.e., in our example it means that those ABox assertions that are made by $V_{x_1}^Q$ and $V_{x_2}^Q$).
4. Hence, to sum up those ideas above, basically we translate δ into δ' as follows:

$$\delta' = \textbf{pick } Q(x_1, x_2). \alpha_{pick}(x_1, x_2) ; \delta'_1 ; \textbf{pick true}. \alpha_{del}()$$

where

- $\text{EFF}(\alpha_{pick}) = \{\text{true} \rightsquigarrow \textbf{add } \{V_{x_1}^Q(x_1), V_{x_2}^Q(x_2), \text{State}(temp)\}\},$
- $\text{EFF}(\alpha_{del}) = \{ \text{true} \rightsquigarrow \textbf{add } \{\text{State}(temp)\},$
 $V_{x_1}^Q(x_1) \rightsquigarrow \textbf{del } \{V_{x_1}^Q(x_1)\},$
 $V_{x_2}^Q(x_2) \rightsquigarrow \textbf{del } \{V_{x_2}^Q(x_2)\} \}$

The idea is that the action α_{pick} stores the information about the picked values for x_1 and x_2 , while α_{del} deletes such information when it is not useful anymore.

5. Note that we need to use the intermediate states (i.e., states marked by $\text{State}(temp)$). Therefore, we also need to translate each atomic action invocation such that it deletes $\text{State}(temp)$ and we also need to translate each formula to be verified such that it ignores the states marked by $\text{State}(temp)$.

Note that such idea also work properly in the case where we have nested pick that ranges over program and might pick a constant for a particular variable more than once. To illustrate the idea, consider the program:

$$\delta = \textbf{pick } Q(x_1, x_2). \delta_1[x_1, x_2]$$

where

$$\delta_1 = \textbf{pick } Q_1(x_1, x_2, x_3). \alpha_1(x_1, x_2, x_3) ; \textbf{pick } Q_2(x_1, x_4). \delta_2[x_1, x_4].$$

and

$$\delta_2 = \textbf{pick } Q_3(x_1, x_4, x_5). \alpha_3(x_1, x_4, x_5).$$

We can then translate δ into δ' as follows:

$$\delta' = \textbf{pick } Q(x_1, x_2). \alpha_{pick}(x_1, x_2) ; \delta'_1 ; \textbf{pick true}. \alpha_{del}()$$

where

- $\text{EFF}(\alpha_{pick}) = \{\text{true} \rightsquigarrow \textbf{add } \{V_{x_1}^Q(x_1), V_{x_2}^Q(x_2), \text{State}(temp)\}\},$

- $\text{EFF}(\alpha_{del}) = \{ \text{true} \rightsquigarrow \mathbf{add} \{ \text{State}(temp) \},$
 $V_{x_1}^Q(x_1) \rightsquigarrow \mathbf{del} \{ V_{x_1}^Q(x_1) \},$
 $V_{x_2}^Q(x_2) \rightsquigarrow \mathbf{del} \{ V_{x_2}^Q(x_2) \} \}$,
- $\delta'_1 = \mathbf{pick} Q_1(x_1, x_2, x_3) \wedge Q^x(x_1, x_2). \alpha_1(x_1, x_2, x_3);$
 $\mathbf{pick} Q_2(x_1, x_4) \wedge Q^x(x_1, x_2). \alpha'_{pick}(x_1, x_4) ; \delta_4 ; \mathbf{pick} \text{true}. \alpha'_{del}(),$ where
- $Q^x(x_1, x_2) = V_{x_1}^Q(x_1) \wedge V_{x_2}^Q(x_2)$
- $\text{EFF}(\alpha'_{pick}) = \{ \text{true} \rightsquigarrow \mathbf{add} \{ V_{x_1}^{Q_2}(x_1), V_{x_4}^{Q_2}(x_4), \text{State}(temp) \} \},$
- $\text{EFF}(\alpha'_{del}) = \{ \text{true} \rightsquigarrow \mathbf{add} \{ \text{State}(temp) \},$
 $V_{x_1}^{Q_2}(x_1) \rightsquigarrow \mathbf{del} \{ V_{x_1}^{Q_2}(x_1) \},$
 $V_{x_4}^{Q_2}(x_4) \rightsquigarrow \mathbf{del} \{ V_{x_4}^{Q_2}(x_4) \} \}$,
- $\delta_4 = \mathbf{pick} Q_3(x_1, x_4, x_5) \wedge Q^x(x_1, x_2) \wedge Q_2^x(x_1, x_4). \alpha_3(x_1, x_4, x_5),$ with
 $* Q_2^x(x_1, x_4) = V_{x_1}^{Q_2}(x_1) \wedge V_{x_4}^{Q_2}(x_4)$

Generalizing the idea above, we now proceed to provide a systematic way to translate GKABs with such extension into our GKABs by defining a translation τ_π that takes as inputs:

1. a program δ (that might contain pick that ranges over a program), and
2. a query Q

and produces a program δ' , in which each pick construct ranges over an action. I.e., we have $\tau_\pi(\delta, Q) = \delta'$. Formally, the translation τ_π is inductively defined over the structure of a program δ as follows:

1. For the case of $\delta = \varepsilon$, we have:

$$\tau_\pi(\varepsilon, Q_\pi(\vec{y})) = \varepsilon,$$

2. For the case of $\delta = \mathbf{pick} Q(\vec{x}). \alpha(\vec{x})$, we have:

$$\tau_\pi(\mathbf{pick} Q(\vec{x}). \alpha(\vec{x}), Q_\pi(\vec{y})) = \mathbf{pick} Q(\vec{x}) \wedge Q_\pi(\vec{y}). \alpha'(\vec{x}),$$

where $\text{EFF}(\alpha') = \text{EFF}(\alpha) \cup \{ \text{true} \rightsquigarrow \mathbf{del} \{ \text{State}(temp) \} \}$

3. For the case of $\delta = \mathbf{pick} Q(\vec{x}). \delta[\vec{x}]$, we have:

$$\tau_\pi(\mathbf{pick} Q(\vec{x}). \delta[\vec{x}], Q_\pi(\vec{y})) = \delta_{pick} ; \tau_\pi(\delta, Q_\pi(\vec{y}) \wedge Q'_\pi(\vec{x})) ; \delta_{del},$$

where

- $\delta_{pick} = \mathbf{pick} Q(\vec{x}) \wedge Q_\pi(\vec{y}). \alpha_{pick}(\vec{x})$, where
 - for each $x \in \vec{x}$, we have $\{ \text{true} \rightsquigarrow \mathbf{add} \{ V_x^Q(x) \} \} \in \text{EFF}(\alpha_{pick})$,
 - V_x^Q is a fresh concept name.
 - $\{ \text{true} \rightsquigarrow \mathbf{add} \{ \text{State}(temp) \} \} \in \text{EFF}(\alpha_{pick})$
- $\delta_{del} = \mathbf{pick} \text{true}. \alpha_{del}()$, where
 - for each $x \in \vec{x}$, we have $\{ V_x^Q(x) \rightsquigarrow \mathbf{del} \{ V_x^Q(x) \} \} \in \text{EFF}(\alpha_{del})$, and
 - $\{ \text{true} \rightsquigarrow \mathbf{add} \{ \text{State}(temp) \} \} \in \text{EFF}(\alpha_{del})$
- $Q'_\pi(\vec{x}) = \bigwedge_{x \in \vec{x}} V_x^Q(x)$

4. For the case of $\delta = \delta_1 | \delta_2$, we have:

$$\tau_\pi(\delta_1 | \delta_2, Q_\pi(\vec{y})) = \tau_\pi(\delta_1, Q_\pi(\vec{y})) \mid \tau_\pi(\delta_2, Q_\pi(\vec{y})),$$

5. For the case of $\delta = \delta_1; \delta_2$, we have:

$$\tau_\pi(\delta_1; \delta_2, Q_\pi(\vec{y})) = \tau_\pi(\delta_1, Q_\pi(\vec{y})) ; \tau_\pi(\delta_2, Q_\pi(\vec{y})),$$

6. For the case of $\delta = \mathbf{if} \varphi \mathbf{then} \delta_1 \mathbf{else} \delta_2$, we have:

$$\tau_\pi(\mathbf{if} \varphi \mathbf{then} \delta_1 \mathbf{else} \delta_2, Q_\pi(\vec{y})) = \mathbf{if} \varphi \mathbf{then} \tau_\pi(\delta_1, Q_\pi(\vec{y})) \mathbf{else} \tau_\pi(\delta_2, Q_\pi(\vec{y}))$$

7. For the case of $\delta = \mathbf{while} \varphi \mathbf{do} \delta$, we have:

$$\tau_\pi(\mathbf{while} \varphi \mathbf{do} \delta, Q_\pi(\vec{y})) = \mathbf{while} \varphi \mathbf{do} \tau_\pi(\delta, Q_\pi(\vec{y}))$$

Hence, given a GKAB $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ where δ might use the pick construct that ranges over a program, we can translate it into a GKAB $\mathcal{G}' = \langle T, A_0, \Gamma, \delta' \rangle$ where $\delta' = \tau_\pi(\delta, \mathbf{true})$ and each pick construct in δ' ranges over a single action.

Concerning Golog procedures, first of all, it is easy to see that GKABs can simulate non-recursive procedures. The idea is to simply unfold each procedure call with its definition, until all procedure calls have been removed from the program. However, it is also possible to show that GKABs can also simulate arbitrary recursive Golog procedures, by explicitly implementing the stack used by the recursion, and making use of loops. This is fully in line with the standard results in computation and programming language theory showing that recursion can be encoded by means of while loops (see, e.g., [136, 7, 122]). The details of this translation still need to be worked out, but one can draw inspiration for this from work that establishes a generic translation from Golog programs (possibly containing recursive procedures) into basic action theories [106]. Notice that, due to the need to explicitly represent and maintain the stack of recursive calls, even when the GKAB is run-bounded, the resulting KAB might not be so.

INCONSISTENCY-AWARE GOLOG-KABs (I-GKABs)

We have seen KABs in Chapter 3 as well as its extension to GKABs in Chapter 4 which provide a sophisticated framework that captures the evolution of KBs by actions. However so far they treat inconsistency in a simplistic way. Basically, they handle inconsistency by rejecting inconsistent states produced through action execution. In general, this is not satisfactory, since the inconsistency may affect just a small portion of the entire KB, and should be treated in a more careful way.

As a motivating example, consider our Example 4.19. Recall that we have a state s_1 containing an ABox

$$A_1 = \{ \text{ReceivedOrder}(\text{chair}), \text{ApprovedOrder}(\text{table}), \text{designedBy}(\text{table}, \text{alice}), \\ \text{Designer}(\text{alice}), \text{hasDesign}(\text{table}, \text{ecodesign}), \\ \text{hasAssemblingLoc}(\text{table}, \text{bolzano}) \}.$$

As explained in Example 4.19, one possible successor of state s_1 is the state s_2 that contains a T -inconsistent ABox A_2 (note that T is specified in Example 2.17), and

$$A_2 = \{ \text{ReceivedOrder}(\text{chair}), \text{designedBy}(\text{table}, \text{alice}), \underline{\text{Designer}(\text{alice})}, \\ \text{hasDesign}(\text{table}, \text{ecodesign}), \underline{\text{hasAssemblingLoc}(\text{table}, \text{bolzano})}, \\ \text{AssembledOrder}(\text{table}), \text{assembledBy}(\text{table}, \text{alice}), \\ \underline{\text{Assembler}(\text{alice})}, \underline{\text{hasAssemblingLoc}(\text{table}, \text{trento})} \}.$$

Basically, the ABox A_2 is obtained from the execution of action `assembleOrders/0`. This action execution introduces the assertions `Assembler(alice)` and `hasAssemblingLoc(table, trento)` that, together with another assertions (see the underlined assertions), cause an inconsistency due to the violation of TBox assertions `Designer \sqsubseteq \neg Assembler` and `(funct hasAssemblingLoc)`. For the violation of `Designer \sqsubseteq \neg Assembler`, notice that the assembler and the designer are assigned from outside of our system (e.g., other department), and the information is brought into the system by calls to external services (i.e., `GETASSEMBLER(table)` and `GETASSEMBLINGLOC(table)`). Hence, we do not have control over the employee task assignment that is done outside of our system. It could also be the case that by the time of assigning the assembler, alice has been moved from the design department into the assembling department, or it could also be the case that alice is replacing her friend who works as an assembler. Thus, in this situation, it might be desirable to repair the inconsistency (e.g., by removing either `Designer(alice)` or `Assembler(alice)`) instead of rejecting the inconsistent state and block the system evolution. For the violation of `(funct hasAssemblingLoc)`, observe that the assertion `hasAssemblingLoc(table, bolzano)` is introduced by the action `prepareOrders/0` and the value for the assembling location is obtained from the service call `ASSIGNASSEMBLINGLOC(table)`. Therefore, it could

be the case that during the preparation phase, the assembling location is assigned to bolzano but in the reality, due to some unpredicted situation, the assembling is done in trento or there is just an error, and we do not have any control over it. Thus, it is also desirable in this situation to repair the inconsistency (e.g., by either removing `hasAssemblingLoc(table, bolzano)` or `hasAssemblingLoc(table, trento)`) instead of rejecting the inconsistent state and block the system.

Starting from all of these observation, here we leverage on the research about instance-level evolution of knowledge bases [182, 96, 103, 54], and, in particular, on the notion of knowledge base repair [132, 133], in order to make GKABs inconsistency-aware. In particular we exploit filter relations in GKABs to define three inconsistency-aware semantics that incorporate the different repair-based approaches in order to handle the inconsistencies.

As in KABs and GKABs, in the following we use *DL-Lite_A* for expressing KBs and we also do not distinguish between objects and values (thus we drop attributes). Moreover we make use of a countably infinite set Δ of constants, which intuitively denotes all possible values in the system. Additionally, we consider a finite set of distinguished constants $\Delta_0 \subset \Delta$, and a finite set \mathcal{F} of *function symbols* that represents *service calls*, which abstractly account for the injection of fresh values (constants) from Δ into the system.

The corresponding publications of the results presented in this chapter are [75, 77, 76, 66, 64, 65].

5.1 Inconsistency Management in DL KBs

We open this chapter by elaborating some inconsistency management approaches in DL KBs. Basically, retrieving certain answers from a KB makes sense only if the KB is consistent: if it is not, then each query returns all possible tuples of constants of the ABox. In a dynamic setting where the ABox evolves over time, consistency is a too strong requirement, and in fact a number of approaches have been proposed to handle the instance-level evolution of KBs, managing inconsistency when it arises. Such approaches typically follow one of the two following two strategies:

1. inconsistencies are kept in the KBs, but the semantics of query answering is refined to take this into account (*consistent query answering* [35]);
2. the extensional part of an inconsistent KB is (minimally) *repaired* so as to remove inconsistencies, and certain answers are then applied over the curated KB.

Here, we follow the approach that focuses on repair-based approaches. We then recall the basic notions related to inconsistency management via repair, distinguishing approaches that repair an ABox and those that repair an update.

5.1.1 ABox repairs

Starting from the seminal work in [96], in [132] two approaches for repairing KBs are proposed: *ABox repair* (AR) and *intersection ABox repair* (IAR). Here we use these approaches to handle inconsistency in KABs, and are respectively called *bold-repair* (*b-repair*) and *certain-repair* (*c-repair*).

Definition 5.1 (Bold-repair). Given an ABox A and a TBox T , a *b-repair* of an ABox A w.r.t. T is an ABox A' such that

1. $A' \subseteq A$,
2. A' is T -consistent, and
3. there does not exist A'' such that $A' \subset A'' \subseteq A$ and A'' is T -consistent.

We also call A' a *maximal T -consistent subset* of A . ■

We denote by $\text{B-REP}(T, A)$ the set of all *b-repairs* of A w.r.t. T .

Definition 5.2 (Certain-repair). Given an ABox A and a TBox T , a *c-repair* of A w.r.t. T is the (unique) set $\text{C-REP}(T, A) = \cap_{A_i \in \text{B-REP}(T, A)} A_i$ of ABox assertions, obtained by intersecting all *b-repairs* of A w.r.t. T . ■

Example 5.3. Continuing Example 4.19, recall that we have an inconsistent ABox A_2 w.r.t. TBox T , where T is specified in Example 2.17, and

$$A_2 = \{ \text{ReceivedOrder}(\text{chair}), \text{designedBy}(\text{table}, \text{alice}), \text{Designer}(\text{alice}), \\ \text{hasDesign}(\text{table}, \text{ecodesign}), \text{hasAssemblingLoc}(\text{table}, \text{bolzano}), \\ \text{AssembledOrder}(\text{table}), \text{assembledBy}(\text{table}, \text{alice}), \\ \text{Assembler}(\text{alice}), \text{hasAssemblingLoc}(\text{table}, \text{trento}) \}.$$

Example of Bold-repair.

In this case, we have $\text{B-REP}(T, A_2) = \{A_2^1, A_2^2, A_2^3, A_2^4\}$, where

$$A_2^1 = \{ \text{ReceivedOrder}(\text{chair}), \text{designedBy}(\text{table}, \text{alice}), \\ \text{hasDesign}(\text{table}, \text{ecodesign}), \text{AssembledOrder}(\text{table}), \\ \text{assembledBy}(\text{table}, \text{alice}), \text{Assembler}(\text{alice}), \\ \text{hasAssemblingLoc}(\text{table}, \text{trento}) \}.$$

$$A_2^2 = \{ \text{ReceivedOrder}(\text{chair}), \text{designedBy}(\text{table}, \text{alice}), \text{Designer}(\text{alice}), \\ \text{hasDesign}(\text{table}, \text{ecodesign}), \text{AssembledOrder}(\text{table}), \\ \text{assembledBy}(\text{table}, \text{alice}), \text{hasAssemblingLoc}(\text{table}, \text{trento}) \}.$$

$$A_2^3 = \{ \text{ReceivedOrder}(\text{chair}), \text{designedBy}(\text{table}, \text{alice}), \\ \text{hasDesign}(\text{table}, \text{ecodesign}), \text{hasAssemblingLoc}(\text{table}, \text{bolzano}), \\ \text{AssembledOrder}(\text{table}), \text{assembledBy}(\text{table}, \text{alice}), \\ \text{Assembler}(\text{alice}) \}.$$

$$A_2^4 = \{ \text{ReceivedOrder}(\text{chair}), \text{designedBy}(\text{table}, \text{alice}), \text{Designer}(\text{alice}), \\ \text{hasDesign}(\text{table}, \text{ecodesign}), \text{hasAssemblingLoc}(\text{table}, \text{bolzano}), \\ \text{AssembledOrder}(\text{table}), \text{assembledBy}(\text{table}, \text{alice}) \}.$$

In this scenario, a plausible justification when we drop $\text{Designer}(\text{alice})$ and keep $\text{Assembler}(\text{alice})$ (i.e., A_2^1 and A_2^3) is that it could be the case that alice is just

moved from the Design Department into the Assembling Department. For the other way around, a possible explanation of dropping `Assembler(alice)` and keeping `Designer(alice)` (i.e., A_2^2 and A_2^4) is because it could be the case that due to some exceptional condition alice is just substituting her friend, but she is still a designer. Moreover, a possible justification of dropping `hasAssemblingLoc(table, bolzano)` and keeping `hasAssemblingLoc(table, trento)` (i.e., A_2^1 and A_2^2) is that it could be the case there are some exceptional conditions (e.g., some disasters) such that the assembling place must be changed from the one that has been planned. On the other hand, a plausible explanation when we drop `hasAssemblingLoc(table, trento)` and keep `hasAssemblingLoc(table, bolzano)` (i.e., A_2^3 and A_2^4) is that there is just a mistake in recording the assembling place but the reality is still aligned as it is planned.

Example of Certain-repair.

Regarding c-repair of A_2 , we have the following

$$\begin{aligned}
 \text{C-REP}(T, A_2) &= \cap_{A_i \in \text{B-REP}(T, A_2)} A_i \\
 &= A_2^1 \cap A_2^2 \cap A_2^3 \cap A_2^4 \\
 &= \{ \text{ReceivedOrder}(\text{chair}), \text{designedBy}(\text{table}, \text{alice}), \\
 &\quad \text{hasDesign}(\text{table}, \text{ecodesign}), \text{AssembledOrder}(\text{table}), \\
 &\quad \text{assembledBy}(\text{table}, \text{alice}) \}.
 \end{aligned}$$

The intuition of this approach is that we only keep those facts that are certainly correct (i.e., do not involve in any inconsistency).

5.1.2 Inconsistency in KB evolution

In a setting where the KB is subject to instance-level evolution, b- and c-repairs are computed agnostically from the updates: each update is committed, and only secondly the obtained ABox is repaired if inconsistent. In [54], a so-called *bold semantics* is proposed to apply the notion of repair to the update itself. Specifically, the bold semantics is defined over a consistent KB $\langle T, A \rangle$ and an instance-level update that comprises two ABoxes F^- and F^+ , respectively containing those assertions that have to be deleted from and then added to A . It is assumed that F^+ is T -consistent, and that new assertions have “priority”: if an inconsistency arises, newly introduced assertions are preferred to those already present in A .

Bold-Evolution

Definition 5.4 (Bold-Evolution of an ABox). Let T be a TBox, A an ABox, F^+ a set of ABox assertions to be added, and F^- a set of ABox assertions to be deleted. An *evolution of an ABox A w.r.t. T by F^+ and F^-* , written $\text{EVOL}(T, A, F^+, F^-)$, is an ABox $A_e = F^+ \cup A'$, where

1. $A' \subseteq (A \setminus F^-)$,
2. $F^+ \cup A'$ is T -consistent, and
3. there does not exist A'' such that $A' \subset A'' \subseteq (A \setminus F^-)$ and $F^+ \cup A''$ is T -consistent.

■

Example 5.5. Consider our Example 4.19. Recall that we have an ABox

$$A_1 = \{ \text{ReceivedOrder}(\text{chair}), \text{ApprovedOrder}(\text{table}), \text{designedBy}(\text{table}, \text{alice}), \\ \text{Designer}(\text{alice}), \text{hasDesign}(\text{table}, \text{ecodesign}), \\ \text{hasAssemblingLoc}(\text{table}, \text{bolzano}) \}.$$

Consider that we have

$$F^+ = \{ \text{AssembledOrder}(\text{table}), \text{assembledBy}(\text{table}, \text{alice}), \text{Assembler}(\text{alice}), \\ \text{hasAssemblingLoc}(\text{table}, \text{trento}) \}$$

$$F^- = \{ \text{ApprovedOrder}(\text{table}) \}$$

we then have the following

$$\text{EVOL}(T, A_1, F^+, F^-) = \{ \text{ReceivedOrder}(\text{chair}), \text{designedBy}(\text{table}, \text{alice}), \\ \text{hasDesign}(\text{table}, \text{ecodesign}), \\ \text{AssembledOrder}(\text{table}), \text{assembledBy}(\text{table}, \text{alice}), \\ \text{Assembler}(\text{alice}), \text{hasAssemblingLoc}(\text{table}, \text{trento}) \}.$$

Since $\text{Assembler}(\text{alice})$ and $\text{hasAssemblingLoc}(\text{table}, \text{trento})$ are new facts, when an inconsistency arises after adding these new facts, we keep these two facts and throw away the other facts that together with these two facts cause an inconsistency.

As an intuition, this repair mechanism assumes that the new facts are more correct or reliable. Thus we keep them and throw the old ones. In this scenario, it could be the case that alice has been moved from the Design Department to the Assembling Department. Therefore, the new fact that alice is an assembler is correct and the fact that alice is a designer is obsolete but the system just have not throw it away. Similarly, regarding the fact that the assembling is performed in trento, it could be the case that due to some reasons, they change the assembling place from the one that has been planned.

5.2 Inconsistency-Aware Semantics for GKAB.

Given a GKAB $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$, we exploit filter relations to define three inconsistency-aware semantics that incorporate the repair-based approaches reviewed in Section 5.1. In particular, we introduce 3 filter relations f_B , f_C , and f_E , as follows.

Definition 5.6 (B-repair Filter f_B). A *B-repair Filter* f_B is a relation that consists of tuples of the form $\langle A, F^+, F^-, A' \rangle$ such that $A' \in \text{B-REP}(T, (A \setminus F^-) \cup F^+)$, where A and A' are ABoxes, and F^+ as well as F^- are two sets of ABox assertions. ■

B-repair Filter f_B

Filter f_B gives rise to the *b-repair execution semantics* for GKABs, where inconsistent ABoxes are repaired by non-deterministically picking a b-repair. Precisely, transition systems which provide the b-repair execution semantics for GKABs is defined as follows.

Definition 5.7 (GKAB B-Transition System). Given a GKAB \mathcal{G} and a b-repair filter f_B , the *b-transition system* of \mathcal{G} , written $\mathcal{Y}_{\mathcal{G}}^{f_B}$, is the transition system of \mathcal{G} w.r.t. f_B . ■

We call *B-GKABs* the GKABs adopting this semantics.

Example 5.8. As an example for B-GKABs, let $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ be a B-GKAB where T , A_0 , Γ , and δ are as in Example 4.3. Similar to Example 4.19, executing \mathcal{G} starting from s_0 , we have $s_1 = \langle A_1, m_1, \delta' \rangle$ as a reachable state from s_0 , where

- $A_1 = \{ \text{ReceivedOrder}(\text{chair}), \text{ApprovedOrder}(\text{table}), \text{designedBy}(\text{table}, \text{alice}), \text{Designer}(\text{alice}), \text{hasDesign}(\text{table}, \text{ecodesign}), \text{hasAssemblingLoc}(\text{table}, \text{bolzano}) \}$,
- $m_1 = \{ [\text{GETDESIGNER}(\text{table}) \rightarrow \text{alice}], [\text{GETDESIGN}(\text{table}) \rightarrow \text{ecodesign}], [\text{ASSIGNASSEMBLINGLOC}(\text{table}) \rightarrow \text{bolzano}] \}$,
- $\delta' = \delta_3; \delta_4; \delta_5; \text{while } [\exists x. \text{Order}(x)] \text{ do } \delta_0$.

As explained in Example 4.19, the next step is to execute δ_3 that is an action invocation of the form **pick true.assembleOrders()** and it involves the service calls GETASSEMBLER/1 and GETASSEMBLINGLOC/1. Thus, it is easy to see that there are infinite successor states of s_1 each of the form $\langle A'_2, m_2, \delta'' \rangle$ where

$$A'_2 \in \text{B-REP}(T, (A_1 \setminus \{\text{ApprovedOrder}(\text{table})\}) \cup \{ \text{AssembledOrder}(\text{table}), \text{assembledBy}(\text{table}, \text{GETASSEMBLER}(\text{table})), \text{Assembler}(\text{GETASSEMBLER}(\text{table})), \text{hasAssemblingLoc}(\text{table}, \text{GETASSEMBLINGLOC}(\text{table})) \})$$

in which $\text{GETASSEMBLINGLOC}(\text{table})$ as well as $\text{GETASSEMBLER}(\text{table})$ are arbitrarily substituted with constants from Δ by a substitution $\theta \in \text{EVAL}(\text{ADD}(T, A_1, \text{assembleOrders}\sigma))$, and $m_2 = m_1 \cup \theta$. Moreover, we have

$$\delta'' = \delta_4; \delta_5; \text{while } \exists x. [\text{Order}(x)] \wedge \neg [\text{DeliveredOrder}(x)] \text{ do } \delta_0.$$

Now, as an example of successor states of s_1 , consider a possible substitution of $\text{GETASSEMBLINGLOC}(\text{table})$ into “trento” and $\text{GETASSEMBLER}(\text{table})$ into “alice” by a particular substitution $\theta \in \text{EVAL}(\text{ADD}(T, A_1, \text{assembleOrders}\sigma))$. We then have states

- $s_2^1 = \langle A_2^1, m_2, \delta'' \rangle$
- $s_2^2 = \langle A_2^2, m_2, \delta'' \rangle$
- $s_2^3 = \langle A_2^3, m_2, \delta'' \rangle$
- $s_2^4 = \langle A_2^4, m_2, \delta'' \rangle$

as some successor states of s_1 where $\text{B-REP}(T, A_2) = \{A_2^1, A_2^2, A_2^3, A_2^4\}$,

$A_2 = \{ \text{ReceivedOrder}(\text{chair}), \text{designedBy}(\text{table}, \text{alice}), \text{Designer}(\text{alice}),$
 $\text{hasDesign}(\text{table}, \text{ecodesign}), \text{hasAssemblingLoc}(\text{table}, \text{bolzano}),$
 $\text{AssembledOrder}(\text{table}), \text{assembledBy}(\text{table}, \text{alice}),$
 $\text{Assembler}(\text{alice}), \text{hasAssemblingLoc}(\text{table}, \text{trento}) \}$,

and $A_2^1, A_2^2, A_2^3, A_2^4$ are the same as in the example of b-repair in Example 5.3.

Now we proceed further to define the c-repair filter as follows.

Definition 5.9 (C-repair Filter f_C). A *C-repair Filter* f_C is a relation that consists of tuples of the form $\langle A, F^+, F^-, A' \rangle$ such that $A' = \text{C-REP}(T, (A \setminus F^-) \cup F^+)$, where A and A' are ABoxes, and F^+ as well as F^- are two sets of ABox assertions. ■

C-repair Filter f_C

Filter f_C gives rise to the *c-repair execution semantics* for GKABs, where inconsistent ABoxes are repaired by computing their unique c-repair. The transition systems which provide the c-repair execution semantics for GKABs is defined as follows.

Definition 5.10 (GKAB C-Transition System). Given a GKAB \mathcal{G} and a c-repair filter f_C , the *c-transition system* of \mathcal{G} , written $\Upsilon_{\mathcal{G}}^{f_C}$, is the transition system of \mathcal{G} w.r.t. f_C . ■

GKAB C-Transition System

We call *C-GKABs* the GKABs adopting this semantics.

Example 5.11. Similar to Example 5.8, let $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ be a C-GKAB where T , A_0 , Γ , and δ are as in Example 4.3. Same as Example 5.8, we have that $s_1 = \langle A_1, m_1, \delta' \rangle$ is a reachable state from the initial state s_0 of \mathcal{G} , where A_1 , m_1 , δ' are the same as in Example 5.8. The next step is to execute δ_3 that is an action invocation of the form **pick** true.assembleOrders() and it involves the service calls GETASSEMBLER/1 and GETASSEMBLINGLOC/1. Thus, it is easy to see that there are infinite successor states of s_1 , each of the form $\langle A'_2, m_2, \delta'' \rangle$ where

$$A'_2 = \text{C-REP}(T, (A_1 \setminus \{\text{ApprovedOrder}(\text{table})\}) \cup \{ \text{AssembledOrder}(\text{table}), \\ \text{assembledBy}(\text{table}, \text{GETASSEMBLER}(\text{table})), \\ \text{Assembler}(\text{GETASSEMBLER}(\text{table})), \\ \text{hasAssemblingLoc}(\text{table}, \text{GETASSEMBLINGLOC}(\text{table})) \})$$

in which $\text{GETASSEMBLINGLOC}(\text{table})$ as well as $\text{GETASSEMBLER}(\text{table})$ are arbitrarily substituted with constants from Δ by a substitution $\theta \in \text{EVAL}(\text{ADD}(T, A_1, \text{assembleOrders}\sigma))$, and $m_2 = m_1 \cup \theta$. Moreover, we have

$$\delta'' = \delta_4; \delta_5; \text{while } \exists x. [\text{Order}(x)] \wedge \neg [\text{DeliveredOrder}(x)] \text{ do } \delta_0.$$

Now, as an example of a successor, consider a possible substitution of $\text{GETASSEMBLINGLOC}(\text{table})$ into “trento” and $\text{GETASSEMBLER}(\text{table})$ into “alice” by

a particular substitution $\theta \in \text{EVAL}(\text{ADD}(T, A_1, \text{assembleOrders}\sigma))$. We then have a state $s_2 = \langle A'_2, m_2, \delta'' \rangle$ where A'_2 is the same as $\text{C-REP}(T, A_2)$ in the example of c-repair in Example 5.3.

Next, we define the evolution filter as follows.

B-evol Filter f_E **Definition 5.12** (B-evol Filter f_E). A *B-evol Filter* f_E is a relation that consists of tuples of the form $\langle A, F^+, F^-, A' \rangle$ such that $A' = \text{EVOL}(T, A, F^+, F^-)$ and F^+ is T -consistent, where A and A' are ABoxes, and F^+ as well as F^- are two sets of ABox assertions. ■

Filter f_E gives rise to the *b-evol execution semantics* for GKABs where for updates leading to inconsistent ABoxes, their unique bold-evolution is computed. The transition systems which provide the b-evol execution semantics for GKABs is defined as follows.

GKAB E-Transition System **Definition 5.13** (GKAB E-Transition System). Given a GKAB \mathcal{G} and a b-evol filter f_E , the *e-transition system* of \mathcal{G} , written $\Upsilon_{\mathcal{G}}^{f_E}$, is the transition system of \mathcal{G} w.r.t. f_E . ■

We call *E-GKABs* the GKABs adopting this semantics. We group these three forms of GKABs (i.e., B-GKABs, C-GKABs, E-GKABs) under the umbrella of *inconsistency-aware GKABs (I-GKABs)*. The definition of $\mu\mathcal{L}_A^{\text{EQL}}$ verification over I-GKABs is usual, i.e., similar to the case of KABs (see also Definition 3.13).

Example 5.14. Similar to Example 5.8 and Example 5.11, let $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ be an E-GKAB where T , A_0 , Γ , and δ are as in Example 4.3. Same as Examples 5.8 and 5.11, we have that $s_1 = \langle A_1, m_1, \delta' \rangle$ is a reachable state from the initial state s_0 of \mathcal{G} , where A_1 , m_1 , δ' are the same as in Examples 5.8 and 5.11. The next step is to execute δ_3 that is an action invocation of the form **pick true.assembleOrders()** and it involves the service calls **GETASSEMBLER/1** and **GETASSEMBLINGLOC/1**. It is easy to see that there are infinite successor states of s_1 , each of the form $\langle A'_2, m_2, \delta'' \rangle$ where $A'_2 = \text{EVOL}(T, A_1, F^+, F^-)$ with

$$A_1 = \{ \text{ReceivedOrder}(\text{chair}), \text{ApprovedOrder}(\text{table}), \text{designedBy}(\text{table}, \text{alice}), \\ \text{Designer}(\text{alice}), \text{hasDesign}(\text{table}, \text{ecodesign}), \\ \text{hasAssemblingLoc}(\text{table}, \text{bolzano}) \}.$$

$$F^+ = \{ \text{AssembledOrder}(\text{table}), \text{assembledBy}(\text{table}, \text{GETASSEMBLER}(\text{table})), \\ \text{Assembler}(\text{GETASSEMBLER}(\text{table})), \\ \text{hasAssemblingLoc}(\text{table}, \text{GETASSEMBLINGLOC}(\text{table})) \}$$

$$F^- = \{ \text{ApprovedOrder}(\text{table}) \}$$

in which `GETASSEMBLINGLOC(table)` as well as `GETASSEMBLER(table)` are arbitrarily substituted with constants from Δ by a substitution $\theta \in \text{EVAL}(\text{ADD}(T, A_1, \text{assembleOrders}\sigma))$, and $m_2 = m_1 \cup \theta$ (note that F^+ and F^- are the set of assertions to be added and deleted by `assembleOrders/0`). Moreover, we have

$$\delta'' = \delta_4; \delta_5; \textbf{while } \exists x. [\text{Order}(x)] \wedge \neg [\text{DeliveredOrder}(x)] \textbf{ do } \delta_0.$$

Now, consider a possible substitution of `GETASSEMBLINGLOC(table)` into “trento” and `GETASSEMBLER(table)` into “alice” by a particular substitution $\theta \in \text{EVAL}(\text{ADD}(T, A_1, \text{assembleOrders}\sigma))$. We then have a state $s_2 = \langle A'_2, m_2, \delta'' \rangle$ where A'_2 is the same as $\text{EVOL}(T, A_1, F^+, F^-)$ in the example of bold-evolution in Example 5.5.

5.3 Compilation of Inconsistency Management

In this section we show that all inconsistency-aware variants of GKABs introduced in Section 5.2 can be reduced to S-GKABs. In particular, we show that verification of $\mu\mathcal{L}_A^{\text{EQL}}$ formulas over I-GKABs can be reduced to the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ over S-GKABs.

Our general strategy is to show that S-GKABs are sufficiently expressive to incorporate the repair-based approaches of Section 5.1, so that an action executed under a certain inconsistency semantics can be compiled into a Golog program that applies the action with the standard semantics, and then explicitly handles the inconsistency, if needed.

For compactness of presentation, we will use some abbreviations for query atoms similar to Definition 2.41 as follows.

Definition 5.15 (Q-UNSAT-ECQ Query Abbreviations). We define several abbreviations for ECQ queries as follows:

*Q-UNSAT-ECQ
Query Abbreviations*

$$\begin{aligned} q_{\text{unsat}}^f((\text{funct } Z), x, y, z) &= [Z(x, y)] \wedge [Z(x, z)] \wedge \neg[y = z]; \\ q_{\text{unsat}}^n(B_1 \sqsubseteq \neg B_2, x) &= [B_1(x)] \wedge [B_2(x)]; \\ q_{\text{unsat}}^n(R_1 \sqsubseteq \neg R_2, x, y) &= [R_1(x, y)] \wedge [R_2(x, y)]; \end{aligned}$$

■

Similar to the FOL query Q_{unsatFOL}^T (as in Definition 2.43), we can define an ECQ Q_{unsatECQ}^T for checking the satisfiability of a $DL\text{-}Lite_A$ KB by making use the query abbreviations in Definition 5.15 above.

Q_{unsatECQ}^T

Theorem 5.16 ([61]). *Given a KB $\langle T, A \rangle$, we have $\text{CERT}(Q_{\text{unsatECQ}}^T, T, A) = \text{true}$ if and only if A is T -inconsistent.* □

From now on, we also use the following abbreviations to compactly express various ABox assertions:

*Abbreviations for
ABox assertion*

- Let B be a basic concept, an ABox assertion $B(c)$ denotes
 - $N(c)$ if $B = N$,

- $P(c, c')$ if $B = \exists P$,
 - $P(c', c)$ if $B = \exists P^-$,
- where ' c ' is a constant;

- Let R be a basic role, an ABox assertion $R(c_1, c_2)$ denotes
 - $P(c_1, c_2)$ if $R = P$,
 - $P(c_2, c_1)$ if $R = P^-$.

As the last preliminaries before we proceed to compile each I-GKABS into S-GKABS, we introduce some notions related to violations of negative inclusion assertion (resp. functionality assertions).

Violation of a
Negative Inclusion
Assertion

Definition 5.17 (Violation of a Negative Inclusion Assertion). Let $\langle T, A \rangle$ be a KB, and $T \models B_1 \sqsubseteq \neg B_2$. We say $B_1 \sqsubseteq \neg B_2$ is *violated* if there exists a constant c such that $\{B_1(c), B_2(c)\} \subseteq A$. In this situation, we also say that $B_1(c)$ (resp. $B_2(c)$) *violates* $B_1 \sqsubseteq \neg B_2$. Similarly for roles. ■

Violation of a
Functionality
Assertion

Definition 5.18 (Violation of a Functionality Assertion). Let $\langle T, A \rangle$ be a KB, and $(\text{funct } R) \in T$. We say $(\text{funct } R)$ is *violated* if there exists constants c, c_1, c_2 such that $\{R(c, c_1), R(c, c_2)\} \subseteq A$ and $c_1 \neq c_2$. In this situation, we also say that $R(c, c_1)$ (resp. $R(c, c_2)$) *violates* $(\text{funct } R)$. ■

For a technical reason, to encode I-GKABS into S-GKABS, we reserve a special ABox assertion $\text{State}(\text{temp})$, where $\text{temp} \in \Delta_0$ and State is a reserved concept name (i.e., outside of any TBox vocabulary). In particular, we use $\text{State}(\text{temp})$ to distinguish *stable* states, where an atomic action can be applied, from intermediate states used by the S-GKABS to (incrementally) remove inconsistent assertions from the ABox. Stable/repair states are marked by the absence/presence of $\text{State}(\text{temp})$. As in Section 4.4, here the ABox assertion $\text{State}(\text{temp})$ is often also called special marker.

5.3.1 From B-GKABS to Standard GKABS

As a preliminary towards defining the translation from B-GKAB to S-KAB, we first define the notion of b-repair actions and b-repair atomic action invocations which in the end will be used to form a b-repair program. The main purpose of the b-repair program is to mimic the computation of b-repair in B-GKAB and thus we can mimic the whole computation in B-GKAB inside S-GKAB.

B-Repair Actions and
Atomic Action
Invocations

Definition 5.19 (B-Repair Actions and Atomic Action Invocations). Given a TBox T we define the set Γ_b^T of *b-repair actions* over T and the set Λ_b^T of *b-repair atomic action invocations* over T as follows:

1. For each functionality assertion $(\text{funct } R) \in T$, we include in Γ_b^T and Λ_b^T respectively:
 - $\alpha_F(x, y) : \{R(x, z) \wedge \neg[z = y] \rightsquigarrow \mathbf{del} \{R(x, z)\}\} \in \Gamma_b^T$, and
 - $\mathbf{pick} \exists z. q_{\text{unsat}}^f((\text{funct } R), x, y, z). \alpha_F(x, y) \in \Lambda_b^T$.

Essentially, the atomic action invocation and action above together repair an inconsistency related to $(\text{funct } R)$ by removing all tuples causing the inconsistency, except one.

2. For each negative concept inclusion $B_1 \sqsubseteq \neg B_2$ such that $T \models B_1 \sqsubseteq \neg B_2$, we include in Γ_b^T and Λ_b^T respectively:

- $\alpha_{B_1}(x) : \{B_1(x) \rightsquigarrow \mathbf{del} \{B_1(x)\}\} \in \Gamma_b^T$,¹ and
- **pick** $q_{\text{unsat}}^n(B_1 \sqsubseteq \neg B_2, x). \alpha_{B_1}(x) \in \Lambda_b^T$.

Basically, the atomic action invocation and action above together repair an inconsistency related to $B_1 \sqsubseteq \neg B_2$ by removing a constant that is both in B_1 and B_2 from B_1 .

3. For each negative role inclusion $R_1 \sqsubseteq \neg R_2$ such that $T \models R_1 \sqsubseteq \neg R_2$, we include in Γ_b^T and Λ_b^T respectively:
 - $\alpha_{R_1}(x, y) : \{R_1(x, y) \rightsquigarrow \mathbf{del} \{R_1(x, y)\}\} \in \Gamma_b^T$, and
 - **pick** $q_{\text{unsat}}^n(R_1 \sqsubseteq \neg R_2, x, y). \alpha_{R_1}(x, y) \in \Lambda_b^T$.

The atomic action invocation and action above repair an inconsistency related to $R_1 \sqsubseteq \neg R_2$ by removing constants that are in both R_1 and R_2 from R_1 . ■

The B-repair program is then defined below by employing the b-repair atomic action invocations as well as the b-repair actions.

Definition 5.20 (B-Repair Program). Given a B-GKAB $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$. Let Γ_b^T be a set of b-repair actions over T , and $\Lambda_b^T = \{a_1, \dots, a_n\}$ be a set of b-repair atomic action invocations over T . We then define the *b-repair program over T* as follows: B-Repair Program

$$\delta_b^T = \mathbf{while} \ Q_{\text{unsatECQ}}^T \ \mathbf{do} \ \delta_r$$

where $\delta_r = a_1 | a_2 | \dots | a_n$. ■

Intuitively, a repair program δ_b^T iterates while the ABox is inconsistent, and at each iteration, non-deterministically picks one of the sources of inconsistency, and removes one or more assertions causing it. Consequently, the loop is guaranteed to terminate, in a state that corresponds to one of the b-repairs of the initial ABox.

We now define a translation function κ_B which basically concatenates each action invocation with a b-repair program in order to simulate the action executions in B-GKABs. Additionally, the translation function κ_B also serves as a one-to-one correspondence (bijection) between the original and the translated program (as well as between the sub-program).

Definition 5.21 (Program Translation κ_B). Given a B-GKAB $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$, we define a *translation* κ_B that translates a program δ into a program δ' inductively as follows: Program Translation
 κ_B

$$\begin{aligned} \kappa_B(\mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p})) &= \mathbf{pick} \ Q(\vec{p}).\alpha'(\vec{p}); \delta_b^T; \mathbf{pick} \ \text{true}.\alpha_{\text{tmp}}^-() \\ \kappa_B(\varepsilon) &= \varepsilon \\ \kappa_B(\delta_1 | \delta_2) &= \kappa_B(\delta_1) | \kappa_B(\delta_2) \\ \kappa_B(\delta_1; \delta_2) &= \kappa_B(\delta_1); \kappa_B(\delta_2) \\ \kappa_B(\mathbf{if} \ \varphi \ \mathbf{then} \ \delta_1 \ \mathbf{else} \ \delta_2) &= \mathbf{if} \ \varphi \ \mathbf{then} \ \kappa_B(\delta_1) \ \mathbf{else} \ \kappa_B(\delta_2) \\ \kappa_B(\mathbf{while} \ \varphi \ \mathbf{do} \ \delta) &= \mathbf{while} \ \varphi \ \mathbf{do} \ \kappa_B(\delta) \end{aligned}$$

where

¹ Note: if $B_1 = \exists P$, then we have that α_{B_1} is of the form $\alpha_{B_1}(x) : \{P(x, y) \rightsquigarrow \mathbf{del} \{P(x, y)\}\} \in \Gamma_b^T$ (Similarly for the case of $B_1 = \exists P^-$).

- $\alpha_{temp}^-() : \{\mathbf{true} \rightsquigarrow \mathbf{del} \{\mathbf{State}(temp)\}\}$,
- δ_b^T is b-repair program over T ,
- α' is an action obtained from $\alpha(\vec{p}) : \{e_1, \dots, e_m\}$ such that we have $\alpha'(\vec{p}) : \{e_1, \dots, e_m, e_{temp}\}$, where

$$e_{temp} = \mathbf{true} \rightsquigarrow \mathbf{add} \{\mathbf{State}(temp)\}.$$

■

Having all of the machinery above, we are ready to define a translation τ_B that, given a B-GKAB, produces an S-GKAB as follows:

*Translation from
B-GKAB to S-GKAB*

Definition 5.22 (Translation from B-GKAB to S-GKAB). We define a translation τ_B that, given a B-GKAB $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$, produces an S-GKAB $\tau_B(\mathcal{G}) = \langle T_p, A_0, \Gamma' \cup \Gamma_b^T \cup \{\alpha_{temp}^-\}, \delta' \rangle$, where

- T_p is the positive inclusion assertions of T (see Definition 2.12),
- Γ' is obtained from Γ such that for each $\alpha \in \Gamma$ of the form $\alpha(\vec{p}) : \{e_1, \dots, e_m\}$, we have $\alpha' \in \Gamma'$ of the form $\alpha'(\vec{p}) : \{e_1, \dots, e_m, e_{temp}\}$, where

$$e_{temp} = \mathbf{true} \rightsquigarrow \mathbf{add} \{\mathbf{State}(temp)\}.$$

- Γ_b^T is a set of b-repair actions over T ,
- α_{temp}^- is an action of the form $\alpha_{temp}^-() : \{\mathbf{true} \rightsquigarrow \mathbf{del} \{\mathbf{State}(temp)\}\}$,
- $\delta' = \kappa_B(\delta)$.

■

In the translation above, only the positive inclusion assertions T_p of the original TBox T are maintained (guaranteeing that the S-GKAB $\tau_B(\mathcal{G})$ never encounters inconsistency). Moreover, the translation of the program κ_B concatenates each original action invocation with a corresponding “repair” phase. Obviously, this means that when an inconsistent ABox is produced, a single transition in B-GKAB \mathcal{G} corresponds to a sequence of transitions in S-GKAB $\tau_B(\mathcal{G})$. Hence, we need to translate the given $\mu\mathcal{L}_A^{\text{EQL}}$ formula Φ to be verified over B-GKAB \mathcal{G} into a corresponding formula over S-GKAB $\tau_B(\mathcal{G})$. This is done by first obtaining formula $\Phi' = \text{NNF}(\Phi)$, where $\text{NNF}(\Phi)$ denotes the *negation normal form* of Φ . Then translating $\text{NNF}(\Phi)$ using the translation t_j as in Definition 4.31. Intuitively, t_j translates every sub-formula of Φ of the form $\langle \rightarrow \rangle \Psi$ becomes $\langle \rightarrow \rangle \mu Z. ((\mathbf{State}(temp) \wedge \langle \rightarrow \rangle Z) \vee (\neg \mathbf{State}(temp) \wedge t_B(\Psi)))$, so as to translate a next-state condition over \mathcal{G} into reachability of the next stable state over $\tau_B(\mathcal{G})$. Similarly for $[-] \Psi$.

We will show later that $\Upsilon_{\mathcal{G}}^{f_B} \models \Phi$ if and only if $\Upsilon_{\tau_B(\mathcal{G})}^{f_S} \models t_j(\Phi)$. Thus, we can show that the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ over B-GKABs can be reduced to the corresponding verification over S-GKABs.

5.3.1.1 Termination and Correctness of B-repair Program

Towards recasting the $\mu\mathcal{L}_A^{\text{EQL}}$ verification over B-GKABs into S-GKABs, in this section we show that the b-repair program is always terminate and produces the same result as the result of b-repair over a knowledge base. To this aim, we first need introduce some preliminaries. As a start, we define the notion of a set of inconsistent ABox assertions as follows.

Definition 5.23 (Set of Inconsistent ABox Assertions). Given a KB $\langle T, A \rangle$, we define the set $\text{INC}(A)$ containing all ABox assertions that participate in the inconsistencies w.r.t. T as the smallest set satisfying the following:

Set of Inconsistent
ABox Assertions

1. For each TBox assertion $B_1 \sqsubseteq \neg B_2$ such that $T \models B_1 \sqsubseteq \neg B_2$, we have $B_1(c) \in \text{INC}(A)$, if $B_1(c) \in A$ and there exists $B_2(c) \in A$.
2. For each TBox assertion $R_1 \sqsubseteq \neg R_2$ such that $T \models R_1 \sqsubseteq \neg R_2$, we have $R_1(c_1, c_2) \in \text{INC}(A)$ if $R_1(c_1, c_2) \in A$ and there exists $R_2(c_1, c_2) \in A$.
3. For each functional assertion $(\text{funct } R) \in T$, we have $R(c_1, c_2) \in \text{INC}(A)$, if $R(c_1, c_2) \in A$ and there exists $R(c_1, c_3) \in A$ such that $c_2 \neq c_3$.

■

Lemma 5.24. *Given a TBox T and an ABox A , we have $|\text{INC}(A)| = 0$ if and only if A is T -consistent.*

Proof. Trivially follows from the definition. Since there is no ABox assertion violating any functionality or negative inclusion assertions. \square

In the following, we show that an execution of b-repair action always reduces the number of ABox assertions that participate in the inconsistency (i.e., $|\text{INC}(A)|$).

Lemma 5.25. *Let $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ be a B-GKAB, $\tau_B(\mathcal{G})$ be an S-GKAB (with transition system $\mathcal{T}_{\tau_B(\mathcal{G})}^{fs}$) obtained from \mathcal{G} through τ_B , and Γ_b^T be the set of b-repair action over T . Consider a T -inconsistent ABox A , a service call map m , an arbitrary b-repair action $\alpha \in \Gamma_b^T$, and a legal parameter assignment σ for α . If $(\langle A, m \rangle, \alpha\sigma, \langle A', m' \rangle) \in \text{TELL}_{fs}$, then $|\text{INC}(A)| > |\text{INC}(A')|$.*

Proof. Intuitively, the correctness of the claim can be seen by observing that each action in the set Γ_b^T of b-repair action over T only removes ABox assertions that participate in an inconsistency. We proof the claim by reasoning over all cases of b-repair actions as follows:

Case 1: *The actions obtained from functionality assertion $(\text{funct } R) \in T_f$.*

Let α_F be such action and has the following form:

$$\alpha_F(x, y) : \{R(x, z) \wedge \neg[z = y] \rightsquigarrow \mathbf{del} \{R(x, z)\}\}.$$

Suppose, α_F is executable in A with a legal parameter assignment σ . Since we have

$$\mathbf{pick} \exists z. q_{\text{unsat}}^f((\text{funct } R), x, y, z). \alpha_F(x, y) \in \Lambda_b^T,$$

then there exists $c \in \text{ADOM}(A)$ and $\{c_1, c_2, c_3, \dots, c_n\} \subseteq \text{ADOM}(A)$ such that $\{R(c, c_1), R(c, c_2), \dots, R(c, c_n)\} \subseteq A$ where $n \geq 2$. W.l.o.g. let σ substitutes x to c , and y to c_1 , then we have $(\langle A, m \rangle, \alpha\sigma, \langle A', m' \rangle) \in \text{TELL}_{fs}$, where $A' = A \setminus \{R(c, c_2), \dots, R(c, c_n)\}$. Therefore we have $|\text{INC}(A)| > |\text{INC}(A')|$.

Case 2: *The actions obtained from negative concept $B_1 \sqsubseteq \neg B_2$ such that $T \models B_1 \sqsubseteq \neg B_2$. Let α_{B_1} be such action and has the following form:*

$$\alpha_{B_1}(x) : \{B_1(x) \rightsquigarrow \mathbf{del} \{B_1(x)\}\}.$$

Suppose, α_{B_1} is executable in A with a legal parameter assignment σ . Since we have

$$\mathbf{pick} q_{\text{unsat}}^n(B_1 \sqsubseteq \neg B_2, x). \alpha_{B_1}(x) \in \Lambda_b^T,$$

then there exists $c \in \text{ADOM}(A)$ such that $\{B_1(c), B_2(c)\} \subseteq A$. W.l.o.g. let σ substitutes x to c , then we have $(\langle A, m \rangle, \alpha\sigma, \langle A', m \rangle) \in \text{TELL}_{f_S}$, where $A' = A \setminus \{B_1(c)\}$. Therefore we have $|\text{INC}(A)| > |\text{INC}(A')|$.

Case 3: The actions obtained from negative role inclusion $R_1 \sqsubseteq \neg R_2$ s.t. $T \models R_1 \sqsubseteq \neg R_2$. The proof is similar to the case 2. \square

As the next preliminaries, in the following we define the notion of a program execution trace as well as the notion when such a trace is called terminating. Moreover, we also define the notion of program execution result in the case of terminating program execution trace.

Program Execution
Trace

Definition 5.26 (Program Execution Trace). Let $\Upsilon_{\mathcal{G}}^f = \langle \Delta, T, \Sigma, s_0, \text{abox}, \Rightarrow \rangle$ be the transition system of a GKAB $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$. Given a state $\langle A_1, m_1, \delta_1 \rangle$, a *program execution trace* π induced by δ on $\langle A_1, m_1, \delta_1 \rangle$ w.r.t. filter f is a (possibly infinite) sequence of states of the form

$$\pi = \langle A_1, m_1, \delta_1 \rangle \rightarrow \langle A_2, m_2, \delta_2 \rangle \rightarrow \langle A_3, m_3, \delta_3 \rangle \rightarrow \dots$$

$$\text{s.t. } \langle A_i, m_i, \delta_i \rangle \xrightarrow{\alpha_i \sigma_i, f} \langle A_{i+1}, m_{i+1}, \delta_{i+1} \rangle \text{ for } i \geq 1. \quad \blacksquare$$

Terminating Program
Execution Trace

Definition 5.27 (Terminating Program Execution Trace). Let $\Upsilon_{\mathcal{G}}^f = \langle \Delta, T, \Sigma, s_0, \text{abox}, \Rightarrow \rangle$ be the transition system of a GKAB $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$. Given a state $\langle A_1, m_1, \delta_1 \rangle$, and a program execution trace π induced by δ_1 on $\langle A_1, m_1, \delta_1 \rangle$, we call π *terminating* if

- (1) $\langle A_1, m_1, \delta_1 \rangle$ is a final state, or
- (2) if $\langle A_1, m_1, \delta_1 \rangle$ is not a final state, then there exists a state $\langle A_n, m_n, \delta_n \rangle$ s.t. we have the following finite program execution trace

$$\pi = \langle A_1, m_1, \delta_1 \rangle \rightarrow \langle A_2, m_2, \delta_2 \rangle \rightarrow \dots \rightarrow \langle A_n, m_n, \delta_n \rangle.$$

where $\langle A_i, m_i, \delta_i \rangle$ (for $i \in \{1, \dots, n-1\}$) are not final states, and $\langle A_n, m_n, \delta_n \rangle$ is a final state.

In the situation (1) (resp. (2)), we call the ABox A_1 (resp. A_n) *the result of executing δ_1 on $\langle A_1, m_1, \delta_1 \rangle$ w.r.t. filter f* . Additionally, we also say that π is the *program execution trace that produces A_1 (resp. A_n)*. \blacksquare

We write $\text{RES}(A_1, m_1, \delta_1)$ to denote the set of all ABoxes that is the result of executing δ_1 on $\langle A_1, m_1, \delta_1 \rangle$ w.r.t. filter f . Note that given a state $\langle A_1, m_1, \delta_1 \rangle$, it is possible to have several terminating program execution traces. Intuitively, a program execution trace is a sequence of states which captures the computation of the program as well as the evolution of the system states by the program. Additionally, it is terminating if at some point it reaches a final state.

We now proceed to show the termination of b-repair program as follows:

Lemma 5.28. Let $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ be a B-GKAB, $\tau_B(\mathcal{G})$ be an S-GKAB (with transition system $\Upsilon_{\tau_B(\mathcal{G})}^{f_S}$) obtained from \mathcal{G} through τ_B , and δ_b^T be a b-repair program over T . We have that δ_b^T is always terminate. I.e., given a state $\langle A, m, \delta_b^T \rangle$ of $\Upsilon_{\tau_B(\mathcal{G})}^{f_S}$, every program execution trace induced by δ_b^T on $\langle A, m, \delta_b^T \rangle$ w.r.t. filter f_S is terminating.

Proof. Roughly speaking, the claim is obtained due to the fact that at each step of the execution of b-repair program δ_b^T , we have that the number of ABox assertions that cause inconsistency are always decreasing (cf. Lemma 5.25). Hence, since the execution of δ_b^T never adds a new ABox assertion and there are only finitely many ABox assertions in the current ABox A , at some point the program δ_b^T will be terminated (when there is no more ABox assertions that cause inconsistency). Technically, we show the claim by dividing the proof into two cases:

Case 1: A is T -consistent.

Trivially true, since $\text{ASK}(Q_{\text{unsatECQ}}^T, T, A) = \text{false}$, we have $\langle A, m, \delta_b^T \rangle$ is a final state, by the definition.

Case 2: A is T -inconsistent.

Given a state $\langle A, m, \delta_b^T \rangle$ such that A is T -inconsistent, w.l.o.g. let

$$\pi = \langle A, m, \delta_b^T \rangle \rightarrow \langle A_1, m, \delta_1 \rangle \rightarrow \langle A_2, m, \delta_2 \rangle \rightarrow \dots$$

be an arbitrary program execution trace induced by δ_b^T on $\langle A, m, \delta_b^T \rangle$ w.r.t. filter f_S . Notice that the service call map m always stay the same since every b-repair action $\alpha \in \Gamma_b^T$ (which is the only action that might appears in δ_b^T) does not involve any service calls. Now, we have to show that eventually there exists a state $\langle A_n, m, \delta_n \rangle$, such that

$$\pi = \langle A, m, \delta_b^T \rangle \rightarrow \langle A_1, m, \delta_1 \rangle \rightarrow \dots \rightarrow \langle A_n, m, \delta_n \rangle$$

and $\langle A_n, m, \delta_n \rangle$ is a final state. By Lemma 5.25, we have that

$$|\text{INC}(A)| > |\text{INC}(A_1)| > |\text{INC}(A_2)| > \dots$$

Additionally, due to the following facts:

- (1) Since we assume that every concepts (resp. roles) are satisfiable, inconsistency can only be caused by
 - a) pair of assertions $B_1(c)$ and $B_2(c)$ (resp. $R_1(c_1, c_2)$ and $R_2(c_1, c_2)$) that violate a negative inclusion assertion $B_1 \sqsubseteq \neg B_2$ (resp. $R_1 \sqsubseteq \neg R_2$) such that $T \models B_1 \sqsubseteq \neg B_2$ (resp. $T \models R_1 \sqsubseteq \neg R_2$), or
 - b) n -number role assertions

$$R(c, c_1), R(c, c_2), \dots, R(c, c_n)$$

that violate a functionality assertion ($\text{funct } R) \in T$.

- (2) To deal with both source of inconsistency in the point (1):
 - a) we consider all negative concept inclusions $B_1 \sqsubseteq \neg B_2$ such that $T \models B_1 \sqsubseteq \neg B_2$ when constructing the b-repair actions Γ_b^T (i.e., we saturate the negative inclusion assertions w.r.t. T obtaining all derivable negative inclusion assertions from T). Moreover, for each negative concept inclusion $B_1 \sqsubseteq \neg B_2$ such that $T \models B_1 \sqsubseteq \neg B_2$, we have an action which removes the ABox assertion $B_1(c)$ (for a certain constant c) in case $B_1 \sqsubseteq \neg B_2$ is violated. Similarly for negative role inclusions.
 - b) we consider all functionality assertions ($\text{funct } R) \in T$ when constructing the b-repair actions Γ_b^T , and each $\alpha_F \in \Gamma_b^T$ removes all role assertions that violates ($\text{funct } R$), except one.

- (3) Observe that $\text{ASK}(Q_{\text{unsatECQ}}^T, T, A_n) = \text{true}$ as long as $|\text{INC}(A)| > 0$ (for any ABox A). Moreover, in such situation, by construction of Λ_b^T , there always exists an executable action $\alpha \in \Gamma_b^T$ (Observe that Q_{unsatECQ}^T is a disjunction of every ECQ Q that guard every corresponding atomic action invocation **pick** $Q(\vec{p}).\alpha(\vec{p}) \in \Lambda_b^T$ of each $\alpha \in \Gamma_b^T$ where each of its free variables are existentially quantified).

As a consequence, eventually there exists A_n such that $|\text{INC}(A_n)| = 0$. Hence by Lemma 5.24 A_n is T -consistent. Therefore $\text{ASK}(Q_{\text{unsatECQ}}^T, T, A_n) = \text{false}$, and $\langle A_n, m, \delta_n \rangle$ is a final state. \square

We now proceed to show the correctness of the b-repair program. I.e., showing that a b-repair program produces exactly the result of a b-repair operation over the given (inconsistent) KB. As the first step, we will show that every ABoxes produced by the b-repair program is a maximal T -consistent subset of the given input ABox as follows.

Lemma 5.29. *Let $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ be a B-GKAB, $\tau_B(\mathcal{G})$ be an S-GKAB (with transition system $\Upsilon_{\tau_B(\mathcal{G})}^{fs}$) obtained from \mathcal{G} through τ_B , and δ_b^T be a b-repair program over T . Consider an ABox A , and a service call map m . if $A' \in \text{RES}(A, m, \delta_b^T)$ then A' is a maximal T -consistent subset of A .*

Proof. Let $A' \in \text{RES}(A, m, \delta_b^T)$. We have to show that

- (1) $A' \subseteq A$
- (2) A' is T -consistent
- (3) There does not exists A'' such that $A' \subset A'' \subseteq A$ and A'' is T -consistent.

We divide the proof into two cases:

Case 1: A is T -consistent. Trivially true, because $\text{ASK}(Q_{\text{unsatECQ}}^T, T, A) = \text{false}$, hence $\langle A, m, \delta_b^T \rangle$ is a final state and $A \in \text{RES}(A, m, \delta_b^T)$. Thus, A trivially satisfies the condition (1) - (3).

Case 2: A is T -inconsistent. Let

$$\pi = \langle A, m, \delta_b^T \rangle \rightarrow \langle A_1, m, \delta_1 \rangle \rightarrow \dots \rightarrow \langle A', m, \delta' \rangle$$

be the corresponding program execution trace that produces A' (This trace should exists because $A' \in \text{RES}(A, m, \delta_b^T)$).

For condition (1). Trivially true from the construction of b-repair program δ_b^T . Since, each step of the program always and only removes some ABox assertions and also by recalling Lemma 5.25 that we have

$$|\text{INC}(A)| > |\text{INC}(A_1)| > |\text{INC}(A_2)| > \dots$$

For condition (2). Since the b-repair program δ_b^T is terminated at a final state $\langle A', m, \delta' \rangle$ where $\text{ASK}(Q_{\text{unsatECQ}}^T, T, A') = \text{false}$, hence A' is T -consistent.

For condition (3). Suppose by contradiction that there exists A'' s.t. $A' \subset A'' \subseteq A$ and A'' is T -consistent. Recall that in $DL\text{-}Lite_A$, since we assume that every concepts (resp. roles) are satisfiable, inconsistency is only caused by

- (i) pair of assertions $B_1(c)$ and $B_2(c)$ (resp. $R_1(c_1, c_2)$ and $R_2(c_1, c_2)$) that violate a negative inclusion assertion $B_1 \sqsubseteq \neg B_2$ (resp. $R_1 \sqsubseteq \neg R_2$) s.t. $T \models B_1 \sqsubseteq \neg B_2$ (resp. $T \models R_1 \sqsubseteq \neg R_2$), or

(ii) n -number role assertions

$$R(c, c_1), R(c, c_2), \dots, R(c, c_n)$$

that violate a functionality assertion ($\text{funct } R \in T$).

However, by the construction of b-repair program δ_b^T , we have that each action $\alpha \in \Gamma_b^T$ is executable when there is a corresponding inconsistency (detected by each guard Q of each corresponding atomic action invocation $\mathbf{pick } Q(\vec{p}).\alpha(\vec{p}) \in \Lambda_b^T$) and each action only either

- (i) removes one of the pair of assertions that violate a negative inclusion assertion, or
- (ii) removes $n - 1$ role assertions among n role assertions that violate a functionality assertion.

Hence, if A'' exists, then there exists an ABox assertion that should not be removed, but then A'' is T -inconsistent. Thus, we have a contradiction. Hence, there does not exist A'' such that $A' \subset A'' \subseteq A$ and A'' is T -consistent. \square

From Lemma 5.29, we can show that every ABox that is produced by b-repair program is in the set of b-repair of the given (inconsistent) KB. Formally it is stated below:

Lemma 5.30. *Let $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ be a B-GKAB, $\tau_B(\mathcal{G})$ be an S-GKAB (with transition system $\Upsilon_{\tau_B(\mathcal{G})}^{fs}$) obtained from \mathcal{G} through τ_B , and δ_b^T be a b-repair program over T . Consider an ABox A and a service call map m . If $A' \in \text{RES}(A, m, \delta_b^T)$ then $A' \in \text{B-REP}(T, A)$.*

Proof. By Lemma 5.29 and the definition of $\text{B-REP}(T, A)$. \square

In order to complete the proof that a b-repair program produces exactly all b-repair results of the given (inconsistent) KB, we will show that every b-repair result of the given (inconsistent) KB is produced by the b-repair program below.

Lemma 5.31. *Let $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ be a B-GKAB, $\tau_B(\mathcal{G})$ be an S-GKAB (with transition system $\Upsilon_{\tau_B(\mathcal{G})}^{fs}$) obtained from \mathcal{G} through τ_B , and δ_b^T be a b-repair program over T . Consider an ABox A and a service call map m . If $A' \in \text{B-REP}(T, A)$, then $A' \in \text{RES}(A, m, \delta_b^T)$.*

Proof. We divide the proof into two cases:

Case 1: A is T -consistent. Trivially true, because $\text{B-REP}(T, A)$ is a singleton set containing A and since $\text{ASK}(Q_{\text{unsatECQ}}^T, T, A) = \text{false}$, we have $\langle A, m, \delta_b^T \rangle$ is a final state and hence $\text{RES}(A, m, \delta_b^T)$ is also a singleton set containing A .

Case 2: A is T -inconsistent. Let A_1 be an arbitrary ABox in $\text{B-REP}(T, A)$, we have to show that there exists $A_2 \in \text{RES}(A, m, \delta_b^T)$ such that $A_2 = A_1$.

Now, consider an arbitrary concept assertion $N(c) \in A_1$ (resp. role assertion $P(c_1, c_2) \in A_1$), we have to show that $N(c) \in A_2$ (resp. $P(c_1, c_2) \in A_2$). For compactness reason, here we only consider the case for $N(c)$ (the case for $P(c_1, c_2)$ is similar). Now we have to consider two cases:

- (a) $N(c)$ does not violate any negative concept inclusion assertion,

- (b) $N(c)$, together with another assertion, violate a negative concept inclusion assertion.

The proof is as follows:

Case (a): It is easy to see that there exists $A_2 \in \text{RES}(A, m, \delta_b^T)$ such that $N(c) \in A_2$ because by construction of δ_b^T , every action $\alpha \in \Gamma_b^T$ never deletes any assertion that does not violate any negative inclusion.

Case (b): Due to the fact about the source of inconsistency in $DL\text{-}Lite_A$, there exists

- i. $N(c) \in A$,
- ii. a negative inclusion $N \sqsubseteq \neg B$ (such that $T \models N \sqsubseteq \neg B$), and
- iii. $B(c) \in A$.

Since $N(c) \in A_1$, then there exists $A'_1 \in \text{B-REP}(T, A)$ such that $B(c) \in A'_1$. Now, it is easy to see from the construction of b-repair program δ_b^T that we have two actions in Γ_b^T that one removes only $N(c)$ from A and the other removes only $B(c)$ from A . Hence, w.l.o.g. we must have $\{A_2, A'_2\} \subseteq \text{RES}(A, m, \delta_b^T)$ such that $N(c) \in A_2$ but $N(c) \notin A'_2$ and $B(c) \notin A_2$ but $B(c) \in A'_2$.

Now, since $N(c)$ is an arbitrary assertion in A , by the two cases above, and also considering that the other case can be treated similarly, we have that $A_2 \in \text{RES}(A, m, \delta_b^T)$, where $A_2 = A_1$. □

As a consequence of Lemmas 5.30 and 5.31, we finally show the correctness of b-repair program (i.e., it produces the same result as the result of b-repair over KB) as follows.

Theorem 5.32. *Let $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ be a B-GKAB, $\tau_B(\mathcal{G})$ be an S-GKAB (with transition system $\Upsilon_{\tau_B(\mathcal{G})}^{fs}$) obtained from \mathcal{G} through τ_B , and δ_b^T be a b-repair program over T . Consider an ABox A and a service call map m , we have that $\text{RES}(A, m, \delta_b^T) = \text{B-REP}(T, A)$.*

Proof. Direct consequence of Lemmas 5.30 and 5.31. □

5.3.1.2 Recasting the Verification of B-GKABs Into S-GKABs

To show that the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ over B-GKABs can be recast as verification over S-GKABs, we make use the J-Bisimulation relation defined in Section 4.4.1 with a slight modification that two J-bisimilar states s_1 and s_2 should have $\text{abox}(s_1) = \text{abox}(s_2)$ instead of $\text{abox}(s_1) \simeq \text{abox}(s_2)$. It is easy to see that Lemmas 4.32 and 4.33 still hold for this small modification. We now aim to show that given a B-GKAB \mathcal{G} , its transition system $\Upsilon_{\mathcal{G}}^{fB}$ is J-bisimilar to the transition system $\Upsilon_{\tau_B(\mathcal{G})}^{fs}$ of S-GKAB $\tau_B(\mathcal{G})$ that is obtained via the translation τ_B . As a consequence, we have that both transition systems $\Upsilon_{\mathcal{G}}^{fB}$ and $\Upsilon_{\tau_B(\mathcal{G})}^{fs}$ can not be distinguished by any $\mu\mathcal{L}_A^{\text{EQL}}$ (in NNF) modulo the translation t_j (as in Definition 4.31).

Lemma 5.33. *Let \mathcal{G} be a B-GKAB with transition system $\Upsilon_{\mathcal{G}}^{fB}$, and let $\tau_B(\mathcal{G})$ be an S-GKAB with transition system $\Upsilon_{\tau_B(\mathcal{G})}^{fs}$ obtained through τ_B . Consider a state $\langle A_b, m_b, \delta_b \rangle$ of $\Upsilon_{\mathcal{G}}^{fB}$ and a state $\langle A_s, m_s, \delta_s \rangle$ of $\Upsilon_{\tau_B(\mathcal{G})}^{fs}$. If $A_s = A_b$, $m_s = m_b$ and $\delta_s = \kappa_B(\delta_b)$, then $\langle A_b, m_b, \delta_b \rangle \sim_J \langle A_s, m_s, \delta_s \rangle$.*

Proof. Let

- $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ and $\Upsilon_{\mathcal{G}}^{f_B} = \langle \Delta, T, \Sigma_b, s_{0b}, abox_b, \Rightarrow_b \rangle$,
- $\tau_B(\mathcal{G}) = \langle T_s, A_0, \Gamma_s, \delta_s \rangle$ and $\Upsilon_{\tau_B(\mathcal{G})}^{f_S} = \langle \Delta, T_s, \Sigma_s, s_{0s}, abox_s, \Rightarrow_s \rangle$.

We have to show the following: for every state $\langle A_b, m_b, \delta_b \rangle \Rightarrow \langle A_b'', m_b'', \delta_b'' \rangle$, there exists states t_1, \dots, t_n , and $\langle A_s'', m_s'', \delta_s'' \rangle$ such that:

- (a) $s \Rightarrow_s t_1 \Rightarrow_s \dots \Rightarrow_s t_n \Rightarrow_s s''$, where $s = \langle A_s, m_s, \delta_s \rangle$, $s'' = \langle A_s'', m_s'', \delta_s'' \rangle$, $n \geq 0$, $\text{State}(\text{temp}) \notin A_s''$, and $\text{State}(\text{temp}) \in abox_s(t_i)$ for $i \in \{1, \dots, n\}$;
- (b) $A_s'' = A_b''$;
- (c) $m_s'' = m_b''$;
- (d) $\delta_s'' = \kappa_B(\delta_b'')$.

By definition of $\Upsilon_{\mathcal{G}}^{f_B}$, Since $\langle A_b, m_b, \delta_b \rangle \Rightarrow \langle A_b'', m_b'', \delta_b'' \rangle$, we have $\langle A_b, m_b, \delta_b \rangle \xrightarrow{\alpha\sigma_b, f_B} \langle A_b'', m_b'', \delta_b'' \rangle$. Hence, by the definition of $\xrightarrow{\alpha\sigma_b, f_B}$, we have:

- $\langle \langle A_b, m_b \rangle, \alpha\sigma_b, \langle A_b'', m_b'' \rangle \rangle \in \text{TELL}_{f_B}$, and
- σ_b is a legal parameter assignment for α in A_b w.r.t. **pick** $Q(\vec{p}).\alpha(\vec{p})$ (i.e., $\text{ASK}(Q\sigma_b, T, A_b) = \text{true}$).

Since $\langle \langle A_b, m_b \rangle, \alpha\sigma_b, \langle A_b'', m_b'' \rangle \rangle \in \text{TELL}_{f_B}$, by the definition of TELL_{f_B} , there exists $\theta_b \in \text{EVAL}(\text{ADD}(T, A_b, \alpha\sigma_b))$ such that

- θ_b and m_b agree on the common values in their domains.
- $m_b'' = m_b \cup \theta_b$.
- $\langle A_b, \text{ADD}(T, A_b, \alpha\sigma_b)\theta_b, \text{DEL}(T, A_b, \alpha\sigma_b), A_b'' \rangle \in f_B$.
- A_b'' is T -consistent.

Since $\langle A_b, \text{ADD}(T, A_b, \alpha\sigma_b)\theta_b, \text{DEL}(T, A_b, \alpha\sigma_b), A_b'' \rangle \in f_B$, by the definition of f_B , there exists A_b' such that $A_b'' \in \text{B-REP}(T, A_b')$, and $A_b' = (A_b \setminus \text{DEL}(T, A_b, \alpha\sigma_b)) \cup \text{ADD}(T, A_b, \alpha\sigma_b)\theta_b$.

Since $\delta_s = \kappa_B(\delta_b)$, by the definition of κ_B , we have that

$$\kappa_B(\text{pick } Q(\vec{p}).\alpha(\vec{p})) = \text{pick } Q(\vec{p}).\alpha'(\vec{p}); \delta_b^T; \text{pick true}.\alpha_{tmp}^-()$$

Hence, the next executable sub-program on state $\langle A_s, m_s, \delta_s \rangle$ is

$$\delta_s' = \text{pick } Q(\vec{p}).\alpha'(\vec{p}); \delta_b^T; \text{pick true}.\alpha_{tmp}^-().$$

Now, since

- $\alpha' \in \Gamma_s$ is obtained from $\alpha \in \Gamma$ through τ_B ,
- the translation τ_B transform α into α' without changing its parameters,
- σ_b maps parameters of $\alpha \in \Gamma$ to constants in $\text{ADOM}(A_b)$, and
- $A_b = A_s$

we can construct σ_s such that $\sigma_s = \sigma_b$. Moreover, we also know that the certain answers computed over A_b are the same to those computed over A_s . Hence, $\alpha' \in \Gamma_s$ is executable in A_s with legal parameter assignment σ_s . Now, since we have $m_s = m_b$, we can construct θ_s such that $\theta_s = \theta_b$. Hence, we have the following:

- θ_s and m_s agree on the common values in their domains.
- $m_s'' = \theta_s \cup m_s = \theta_b \cup m_b = m_b''$.

Let $A_s' = (A_s \setminus \text{DEL}(T_s, A_s, \alpha'\sigma_s)) \cup \text{ADD}(T_s, A_s, \alpha'\sigma_s)\theta_s$, as a consequence, we have that $\langle A_s, \text{ADD}(T_s, A_s, \alpha'\sigma_s)\theta_s, \text{DEL}(T_s, A_s, \alpha'\sigma_s), A_s' \rangle \in f_S$. Since $A_s = A_b$, $\sigma_s = \sigma_b$ and $\theta_s = \theta_b$, it follows that

- $\text{DEL}(T_s, A_s, \alpha'\sigma_s) = \text{DEL}(T, A_b, \alpha\sigma_b)$, and

- $\text{ADD}(T_s, A_s, \alpha' \sigma_s) \theta_s \setminus \text{State}(\text{temp}) = \text{ADD}(T, A_b, \alpha \sigma_b) \theta_b$.

Hence, by the construction of A'_s and A'_b above, we have $A'_b = A'_s \setminus \text{State}(\text{temp})$. By the definition of τ_B , we have $T_s = T_p$ (i.e., only positive inclusion assertion of T), hence A'_s is T_s -consistent. Thus, by the definition of TELL_{f_s} , we have $\langle \langle A_s, m_s \rangle, \alpha' \sigma_s, \langle A'_s, m''_s \rangle \rangle \in \text{TELL}_{f_s}$. Moreover, we have

$$\langle A_s, m_s, \text{pick } Q(\vec{p}).\alpha'(\vec{p}); \delta_0 \rangle \xrightarrow{\alpha' \sigma_s, f_s} \langle A'_s, m''_s, \delta_0 \rangle$$

where $\delta_0 = \delta_b^T; \text{pick true}.\alpha_{tmp}^-()$.

Now, we need to show that the rest of program in δ'_s that still need to be executed (i.e., δ_0) will bring us into a state $\langle A''_s, m''_s, \delta''_s \rangle$ s.t. the claim **(a)** - **(e)** are proved. Since δ_0 does not involve any service calls, w.l.o.g. let

$$\pi = \langle A'_s, m''_s, \delta_0 \rangle \rightarrow \langle A_1, m''_s, \delta_1 \rangle \rightarrow \dots$$

be a program execution trace induced by δ_0 on $\langle A'_s, m''_s, \delta_0 \rangle$. By Lemma 5.28 and Theorem 5.32, we have that

- δ_b^T is always terminate,
 - δ_b^T produces an ABox A_n such that $A_n \in \text{B-REP}(T, A'_s)$,
- additionally, by the construction of δ_b^T and α_{tmp}^- , we have that
- δ_b^T never deletes $\text{State}(\text{temp})$, and
 - α_{tmp}^- only deletes $\text{State}(\text{temp})$ from the corresponding ABox,

therefore, there exists a state $\langle A''_s, m''_s, \delta_{n+1} \rangle$ such that we have the following program execution trace

$$\pi = \langle A'_s, m''_s, \delta_0 \rangle \rightarrow \langle A_1, m''_s, \delta_1 \rangle \rightarrow \dots \rightarrow \langle A_n, m''_s, \delta_n \rangle \rightarrow \langle A''_s, m''_s, \delta_{n+1} \rangle$$

where

- $\text{State}(\text{temp}) \notin A''_s$,
- $\text{State}(\text{temp}) \in A'_s$, $\text{State}(\text{temp}) \in A_i$ (for $1 \leq i \leq n$),
- $\langle A''_s, m''_s, \delta_{n+1} \rangle$ is a final state,
- $A_n \in \text{B-REP}(T, A'_s)$ (by Theorem 5.32),
- $A''_s \in \text{B-REP}(T, A'_b)$ (Because $A'_b = A'_s \setminus \text{State}(\text{temp})$, $A''_s = A_n \setminus \text{State}(\text{temp})$, $A_n \in \text{B-REP}(T, A'_s)$, and $\text{State}(\text{temp})$ is a special marker).

Since $\langle A''_s, m''_s, \delta_{n+1} \rangle$ is a final state, we have finished executing δ'_s , and by the definition of κ_B the rest of the program to be executed is $\delta''_s = \kappa_B(\delta''_b)$.

Therefore, we have shown that there exists s'', t_1, \dots, t_n (for $n \geq 0$) such that

$$s \Rightarrow_s t_1 \Rightarrow_s \dots \Rightarrow_s t_n \Rightarrow_s s''$$

where

- $s = \langle A_s, m_s, \delta_s \rangle$, $s'' = \langle A''_s, m''_s, \delta''_s \rangle$,
- $\text{State}(\text{temp}) \notin A''_s$, and
- $\text{State}(\text{temp}) \in \text{abox}_2(t_i)$ for $i \in \{1, \dots, n\}$;
- $A''_s = A'_b$

The other direction of bisimulation relation can be proven similarly. □

Having Lemma 5.33 in hand, we can easily show that given a B-GKAB \mathcal{G} , its transition system $\Upsilon_{\mathcal{G}}^{f_B}$ is J-bisimilar to the transition $\Upsilon_{\tau_B(\mathcal{G})}^{f_S}$ of S-GKAB $\tau_B(\mathcal{G})$ (which is obtained via the translation τ_B).

Lemma 5.34. *Given a B-GKAB \mathcal{G} , we have $\Upsilon_{\mathcal{G}}^{fB} \sim_J \Upsilon_{\tau_B(\mathcal{G})}^{fS}$*

Proof. Let

1. $\mathcal{G} = \langle T, A_0, \Gamma, \delta_b \rangle$ and $\Upsilon_{\mathcal{G}}^{fB} = \langle \Delta, T, \Sigma_b, s_{0b}, abox_b, \Rightarrow_b \rangle$,
2. $\tau_B(\mathcal{G}) = \langle T_s, A_0, \Gamma_s, \delta_s \rangle$, and $\Upsilon_{\tau_B(\mathcal{G})}^{fS} = \langle \Delta, T_s, \Sigma_s, s_{0s}, abox_s, \Rightarrow_s \rangle$.

We have that $s_{0b} = \langle A_0, m_b, \delta_b \rangle$ and $s_{0s} = \langle A_0, m_s, \delta_s \rangle$ where $m_b = m_s = \emptyset$. By the definition of τ_B , we also have $\delta_s = \kappa_B(\delta_b)$. Hence, by Lemma 5.33, we have $s_{0b} \sim_J s_{0s}$. Therefore, by the definition of J-bisimulation, we have $\Upsilon_{\mathcal{G}}^{fB} \sim_J \Upsilon_{\tau_B(\mathcal{G})}^{fS}$. \square

With all of these machinery in hand, we are now ready to show that the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ over B-GKABs can be recast as verification over S-GKAB as follows.

Theorem 5.35. *Given a B-GKAB \mathcal{G} and a closed $\mu\mathcal{L}_A^{\text{EQL}}$ formula Φ in NNF,*

$$\Upsilon_{\mathcal{G}}^{fB} \models \Phi \text{ iff } \Upsilon_{\tau_B(\mathcal{G})}^{fS} \models t_j(\Phi)$$

Proof. By Lemma 5.34, we have that $\Upsilon_{\mathcal{G}}^{fB} \sim_J \Upsilon_{\tau_B(\mathcal{G})}^{fS}$. Hence, the claim is directly follows from Lemma 4.33. \square

5.3.2 From C-GKABs to Standard GKABs

Making inconsistency management for C-GKABs explicit requires just a single action, which removes all ABox assertions that are involved in some form of inconsistency. To this aim, we define a 0-ary c-repair action α_c^T , as follows:

Definition 5.36 (C-Repair Action). Given a TBox T , we define a 0-ary (i.e., has no action parameters) *c-repair action* α_c^T over T , where $\text{EFF}(\alpha_c^T)$ is the smallest set containing the following effects:

C-Repair Action

- for each functionality assertion $(\text{funct } R) \in T$, we have
$$q_{\text{unsat}}^f((\text{funct } R), x, y, z) \rightsquigarrow \mathbf{del} \{R(x, y), R(x, z)\} \in \text{EFF}(\alpha_c^T)$$
- for each negative concept inclusion assertion $B_1 \sqsubseteq \neg B_2$ such that $T \models B_1 \sqsubseteq \neg B_2$, we have
$$q_{\text{unsat}}^n(B_1 \sqsubseteq \neg B_2, x) \rightsquigarrow \mathbf{del} \{B_1(x), B_2(x)\} \in \text{EFF}(\alpha_c^T);^2$$
- for each negative role inclusion assertion $R_1 \sqsubseteq \neg R_2$ such that $T \models R_1 \sqsubseteq \neg R_2$, we have
$$q_{\text{unsat}}^n(R_1 \sqsubseteq \neg R_2, x, y) \rightsquigarrow \mathbf{del} \{R_1(x, y), R_2(x, y)\} \in \text{EFF}(\alpha_c^T).$$
- $\text{true} \rightsquigarrow \mathbf{del} \{\text{State}(\text{temp})\} \in \text{EFF}(\alpha_c^T)$.

■

Roughly speaking, the c-repair action deletes every ABox assertion that involves in inconsistency. Technically, the first three effects of the c-repair action above will delete the ABox assertions that are obtained by populating the atoms in the right hand side

² Note: if $B_1 = \exists P_1$ (resp. $B_2 = \exists P_2$), then with a little abuse of notation, specifically for this case, we have that the atom $B_1(x)$ (resp. $B_2(x)$) denotes $P_1(x, y)$ (resp. $P_2(x, y)$). For instance, for the assertion $\exists P_1 \sqsubseteq \neg N$, we have $P_1(x, y) \wedge N(x) \rightsquigarrow \mathbf{del} \{P_1(x, y), N(x)\} \in \text{EFF}(\alpha_c^T)$. Similarly when $B_1 = \exists P_1^-$ (resp. $B_2 = \exists P_2^-$).

of the effects with every constant that satisfies the unsatisfiability query in the left hand side of the effects. Notice that all effects are guarded by queries that extract only constants involved in an inconsistency. Hence, other assertions are kept unaltered, which also means that α_c^T is a no-op when applied over a T -consistent ABox. The effect in the last line above is used to flush the marker for an intermediate state.

We now define a translation function κ_C that essentially concatenates each action invocation with a c-repair action in order to simulate the action executions in C-GKABs. Additionally, the translation function κ_C also serves as a one-to-one correspondence (bijection) between the original and the translated program (as well as between the sub-program).

Program Translation
 κ_C

Definition 5.37 (Program Translation κ_C). Given a C-GKAB $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$, we define a *translation* κ_C that translates a program δ into a program δ' inductively as follows:

$$\begin{aligned} \kappa_C(\mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p})) &= \mathbf{pick} \ Q(\vec{p}).\alpha'(\vec{p}); \mathbf{pick} \ \mathbf{true}.\alpha_c^T() \\ \kappa_C(\varepsilon) &= \varepsilon \\ \kappa_C(\delta_1|\delta_2) &= \kappa_C(\delta_1)|\kappa_C(\delta_2) \\ \kappa_C(\delta_1;\delta_2) &= \kappa_C(\delta_1);\kappa_C(\delta_2) \\ \kappa_C(\mathbf{if} \ \varphi \ \mathbf{then} \ \delta_1 \ \mathbf{else} \ \delta_2) &= \mathbf{if} \ \varphi \ \mathbf{then} \ \kappa_C(\delta_1) \ \mathbf{else} \ \kappa_C(\delta_2) \\ \kappa_C(\mathbf{while} \ \varphi \ \mathbf{do} \ \delta) &= \mathbf{while} \ \varphi \ \mathbf{do} \ \kappa_C(\delta) \end{aligned}$$

where

- action α' is obtained from $\alpha \in \Gamma$, such that

$$\text{EFF}(\alpha') = \text{EFF}(\alpha) \cup \{\mathbf{true} \rightsquigarrow \mathbf{add} \ \{\text{State}(\text{temp})\}\}$$

- α_c^T is a c-repair action over T .

■

To transform a C-GKAB into the corresponding S-GKAB, We define a translation τ_C that, given a C-GKAB, generates an S-GKAB as follows.

Translation from
C-GKAB to S-GKAB

Definition 5.38 (Translation from C-GKAB to S-GKAB). We define a translation τ_C that, given a C-GKAB $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$, generates an S-GKAB $\tau_C(\mathcal{G}) = \langle T_p, A_0, \Gamma' \cup \{\alpha_c^T\}, \delta' \rangle$, where

- T_p is the positive inclusion assertions of T (see Definition 2.12),
- Γ' is obtained from Γ such that for each $\alpha \in \Gamma$, we have $\alpha' \in \Gamma'$ and

$$\text{EFF}(\alpha') = \text{EFF}(\alpha) \cup \{\mathbf{true} \rightsquigarrow \mathbf{add} \ \{\text{State}(\text{temp})\}\}$$

- α_c^T is a c-repair action over T ,
- $\delta' = \kappa_C(\delta)$.

■

As for B-GKABs, the translation above only maintains positive inclusion assertions of TBox T . Moreover, intuitively the new program obtained from the translation above attests that each transition in C-GKAB \mathcal{G} corresponds to a sequence of two transitions in S-GKAB $\tau_C(\mathcal{G})$: the first mimics the action execution, while the second computes the c-repair of the obtained ABox.

A $\mu\mathcal{L}_A^{\text{EQL}}$ property Φ over C-GKAB \mathcal{G} can then be recast as a corresponding property over $\tau_C(\mathcal{G})$ that simply substitutes each subformula $\langle - \rangle \Psi$ of Φ with $\langle - \rangle \langle - \rangle \Psi$ (similarly for $[-] \Phi$). Formally we define such formula translation as follows:

Definition 5.39 (Translation t_{dup}). We define a *translation* t_{dup} that takes a $\mu\mathcal{L}_A^{\text{EQL}}$ formula Φ as an input and produces a new $\mu\mathcal{L}_A^{\text{EQL}}$ formula $t_{\text{dup}}(\Phi)$ by recurring over the structure of Φ as follows:

Translation t_{dup}

- $t_{\text{dup}}(Q) = Q$
- $t_{\text{dup}}(\neg\Phi) = \neg t_{\text{dup}}(\Phi)$
- $t_{\text{dup}}(\exists x.\Phi) = \exists x.t_{\text{dup}}(\Phi)$
- $t_{\text{dup}}(\Phi_1 \vee \Phi_2) = t_{\text{dup}}(\Phi_1) \vee t_{\text{dup}}(\Phi_2)$
- $t_{\text{dup}}(\mu Z.\Phi) = \mu Z.t_{\text{dup}}(\Phi)$
- $t_{\text{dup}}(\langle - \rangle \Phi) = \langle - \rangle \langle - \rangle t_{\text{dup}}(\Phi)$

■

With these two translations at hand, we will show later that $\mathcal{Y}_{\mathcal{G}}^{fc} \models \Phi$ if and only if $\mathcal{Y}_{\tau_C(\mathcal{G})}^{fs} \models t_{\text{dup}}(\Phi)$ which consequently means that the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ over C-GKABs can be reduced to the corresponding verification over S-GKABs. The core idea of the proof is to use a certain bisimulation relation in which two bisimilar transition systems can not be distinguished by $\mu\mathcal{L}_A^{\text{EQL}}$ properties modulo the formula translation t_{dup} . Then, we show that the transition system of a C-GKAB is bisimilar to the transition system of its corresponding S-GKAB w.r.t. this bisimulation relation.

5.3.2.1 Skip-one Bisimulation (S-Bisimulation)

As a start towards reducing the verification of C-GKABs into S-GKABs, we define the notion of *skip-one bisimulation* as follows.

Definition 5.40 (Skip-one Bisimulation (S-Bisimulation)).

Skip-one Bisimulation (S-Bisimulation)

Let $\mathcal{T}_1 = \langle \Delta, T, \Sigma_1, s_{01}, \text{abox}_1, \Rightarrow_1 \rangle$ and $\mathcal{T}_2 = \langle \Delta, T, \Sigma_2, s_{02}, \text{abox}_2, \Rightarrow_2 \rangle$ be KB transition systems, with $\text{ADOM}(\text{abox}_1(s_{01})) \subseteq \Delta$ and $\text{ADOM}(\text{abox}_2(s_{02})) \subseteq \Delta$. A *skip-one bisimulation* (S-Bisimulation) between \mathcal{T}_1 and \mathcal{T}_2 is a relation $\mathcal{B} \subseteq \Sigma_1 \times \Sigma_2$ such that $\langle s_1, s_2 \rangle \in \mathcal{B}$ implies that:

1. $\text{abox}_1(s_1) = \text{abox}_2(s_2)$
2. for each s'_1 , if $s_1 \Rightarrow_1 s'_1$ then there exists t , and s'_2 with

$$s_2 \Rightarrow_2 t \Rightarrow_2 s'_2$$

such that $\langle s'_1, s'_2 \rangle \in \mathcal{B}$, $\text{State}(\text{temp}) \notin \text{abox}_2(s'_2)$ and $\text{State}(\text{temp}) \in \text{abox}_2(t)$.

3. for each s'_2 , if

$$s_2 \Rightarrow_2 t \Rightarrow_2 s'_2$$

with $\text{State}(\text{temp}) \in \text{abox}_2(t)$ and $\text{State}(\text{temp}) \notin \text{abox}_2(s'_2)$, then there exists s'_1 with $s_1 \Rightarrow_1 s'_1$, such that $\langle s'_1, s'_2 \rangle \in \mathcal{B}$.

■

Let $\mathcal{T}_1 = \langle \Delta, T, \Sigma_1, s_{01}, \text{abox}_1, \Rightarrow_1 \rangle$ and $\mathcal{T}_2 = \langle \Delta, T, \Sigma_2, s_{02}, \text{abox}_2, \Rightarrow_2 \rangle$ be KB transition systems, a state $s_1 \in \Sigma_1$ is *S-bisimilar* to $s_2 \in \Sigma_2$, written $s_1 \sim_{\text{so}} s_2$, if there exists an S-bisimulation relation \mathcal{B} between \mathcal{T}_1 and \mathcal{T}_2 such that $\langle s_1, s_2 \rangle \in \mathcal{B}$. A transition

system \mathcal{T}_1 is *S-bisimilar* to \mathcal{T}_2 , written $\mathcal{T}_1 \sim_{\text{so}} \mathcal{T}_2$, if there exists an S-bisimulation relation \mathcal{B} between \mathcal{T}_1 and \mathcal{T}_2 such that $\langle s_{01}, s_{02} \rangle \in \mathcal{B}$.

Now, we advance further to show that two S-bisimilar transition systems can not be distinguished by any $\mu\mathcal{L}_A^{\text{EQL}}$ formula modulo the translation t_{dup} .

Lemma 5.41. *Consider two KB transition systems $\mathcal{T}_1 = \langle \Delta, T, \Sigma_1, s_{01}, \text{abox}_1, \Rightarrow_1 \rangle$ and $\mathcal{T}_2 = \langle \Delta, T, \Sigma_2, s_{02}, \text{abox}_2, \Rightarrow_2 \rangle$, two states $s_1 \in \Sigma_1$ and $s_2 \in \Sigma_2$ such that $s_1 \sim_{\text{so}} s_2$. Then for every formula Φ of $\mu\mathcal{L}_A^{\text{EQL}}$, and every valuations v_1 and v_2 that assign to each of its free variables a constant $c_1 \in \text{ADOM}(\text{abox}_1(s_1))$ and $c_2 \in \text{ADOM}(\text{abox}_2(s_2))$, such that $c_1 = c_2$, we have that*

$$\mathcal{T}_1, s_1 \models \Phi v_1 \text{ if and only if } \mathcal{T}_2, s_2 \models t_{\text{dup}}(\Phi) v_2.$$

Proof. Similar to Lemma 4.32, we divide the proof into three parts:

- (1) First, we obtain the proof of the claim for formulae of $\mathcal{L}_A^{\text{EQL}}$
- (2) Second, we extend the results to the infinitary logic obtained by extending $\mathcal{L}_A^{\text{EQL}}$ with arbitrary countable disjunction.
- (3) Last, we recall that fixpoints can be translated into this infinitary logic, thus proving that the theorem holds for $\mu\mathcal{L}_A^{\text{EQL}}$.

Since the step (2) and (3) are similar to the proof of Lemma 4.32, here we only highlight some interesting cases of the proof for the step (1) as follow (the other cases of step (1) can be shown similarly):

Proof for $\mathcal{L}_A^{\text{EQL}}$.

Base case:

$(\Phi = Q)$. Since $s_1 \sim_{\text{so}} s_2$, we have $\text{abox}_1(s_1) = \text{abox}_2(s_2)$, and hence $\text{CERT}(Q, T, \text{abox}_1(s_1)) = \text{CERT}(Q, T, \text{abox}_2(s_2))$. Since $t_j(Q) = Q$, for every valuations v_1 and v_2 that assign to each of its free variables a constant $c_1 \in \text{ADOM}(\text{abox}_1(s_1))$ and $c_2 \in \text{ADOM}(\text{abox}_2(s_2))$, such that $c_1 = c_2$, we have

$$\mathcal{T}_1, s_1 \models Q v_1 \text{ if and only if } \mathcal{T}_2, s_2 \models t_{\text{dup}}(Q) v_2.$$

Inductive step:

- $(\Phi = \neg\Psi)$. By Induction hypothesis, for every valuations v_1 and v_2 that assign to each of its free variables a constant $c_1 \in \text{ADOM}(\text{abox}_1(s_1))$ and $c_2 \in \text{ADOM}(\text{abox}_2(s_2))$, such that $c_2 = c_1$, we have that $\mathcal{T}_1, s_1 \models \Psi v_1$ if and only if $\mathcal{T}_2, s_2 \models t_{\text{dup}}(\Psi) v_1$. Hence, $\mathcal{T}_1, s_1 \not\models \Psi v_1$ if and only if $\mathcal{T}_2, s_2 \not\models t_{\text{dup}}(\Psi) v_2$. By definition, $\mathcal{T}_1, s_1 \models \neg\Psi v_1$ if and only if $\mathcal{T}_2, s_2 \models \neg t_{\text{dup}}(\Psi) v_2$. Hence, by the definition of t_{dup} , we have $\mathcal{T}_1, s_1 \models \neg\Psi v_1$ if and only if $\mathcal{T}_2, s_2 \models t_{\text{dup}}(\neg\Psi) v_2$.
- $(\Phi = \langle \rightarrow \rangle \Psi)$. Assume $\mathcal{T}_1, s_1 \models \langle \rightarrow \rangle \Psi v_1$, then there exists s'_1 s.t. $s_1 \Rightarrow_1 s'_1$ and $\mathcal{T}_1, s'_1 \models \Psi v_1$. Since $s_1 \sim_{\text{so}} s_2$, there exist t and s'_2 s.t.

$$s_2 \Rightarrow_2 t \Rightarrow_2 s'_2$$

and $s'_1 \sim_{\text{so}} s'_2$. Hence, by induction hypothesis, for every valuations v_2 that assign to each free variables x of $t_{\text{dup}}(\Psi)$ a constant $c_2 \in \text{ADOM}(\text{abox}_2(s_2))$, such that $c_2 = c_1$ with $x/c_1 \in v_1$, we have

$$\mathcal{T}_2, s'_2 \models t_{\text{dup}}(\Psi) v_2.$$

Since $s_2 \Rightarrow_2 t \Rightarrow_2 s'_2$, therefore we get

$$\mathcal{I}_2, s_2 \models (\langle \neg \rangle \langle \neg \rangle t_{dup}(\Psi))v_2.$$

Since $t_{dup}(\langle \neg \rangle \Phi) = \langle \neg \rangle \langle \neg \rangle t_{dup}(\Phi)$, we therefore have

$$\mathcal{I}_2, s_2 \models t_{dup}(\langle \neg \rangle \Psi)v_2.$$

The other direction can be shown in a similar way. □

Having Lemma 5.41 in hand, we can easily show that two S-bisimilar transition systems can not be distinguished by any $\mu\mathcal{L}_A^{\text{EQL}}$ formulas modulo translation t_{dup} .

Lemma 5.42. *Consider two transition systems \mathcal{I}_1 and \mathcal{I}_2 such that $\mathcal{I}_1 \sim_{\text{so}} \mathcal{I}_2$. For every closed $\mu\mathcal{L}_A^{\text{EQL}}$ formula Φ , we have:*

$$\mathcal{I}_1 \models \Phi \text{ if and only if } \mathcal{I}_2 \models t_{dup}(\Phi)$$

Proof. Let $\mathcal{I}_1 = \langle \Delta_1, T, \Sigma_1, s_{01}, abox_1, \Rightarrow_1 \rangle$ and $\mathcal{I}_2 = \langle \Delta_2, T, \Sigma_2, s_{02}, abox_2, \Rightarrow_2 \rangle$. By the definition of S-bisimilar transition system we have that $s_{01} \sim_{\text{so}} s_{02}$. Thus we obtain the proof as a consequence of Lemma 5.41, due to the fact that

$$\mathcal{I}_1, s_{01} \models \Phi \text{ if and only if } \mathcal{I}_2, s_{02} \models t_{dup}(\Phi)$$
□

5.3.2.2 Important Properties of C-Repair and C-Repair Actions.

To the aim of reducing verification of C-GKABs into S-GKABs, we now aiming to show an important property of c-repair and c-repair action, namely we show that c-repair action produces the same results as the computation of c-repair. To this aim, we first show some important properties of b-repair, c-repair and also c-repair action. As a start, below we show that for every pair of ABox assertions that violates a certain negative inclusion assertion, each of them will be contained in two different ABoxes in the result of b-repair.

Lemma 5.43. *Let T be a TBox, and A be an ABox. For every negative concept inclusion assertion $B_1 \sqsubseteq \neg B_2$ such that $T \models B_1 \sqsubseteq \neg B_2$ and $B_1 \neq B_2$, if $\{B_1(c), B_2(c)\} \subseteq A$ (for any constant $c \in \Delta$), then there exist $A' \in \text{B-REP}(T, A)$ such that (i) $B_1(c) \in A'$, (ii) $B_2(c) \notin A'$. (Similarly for the case of negative role inclusion assertion $R_1 \sqsubseteq \neg R_2$ s.t. $T \models R_1 \sqsubseteq \neg R_2$).*

Proof. Suppose by contradiction $\{B_1(c), B_2(c)\} \subseteq A$, and there does not exist $A' \in \text{B-REP}(T, A)$ such that $B_1(c) \in A'$ and $B_2(c) \notin A'$. Since in $DL\text{-}Lite_A$ the violation of negative concept inclusion $B_1 \sqsubseteq \neg B_2$ is only caused by a pair of assertions $B_1(c)$ and $B_2(c)$ (for any constant $c \in \Delta$) and by the definition of $\text{B-REP}(T, A)$, it contains all maximal T -consistent subsets of A , then there should be a T -consistent ABox $A' \in \text{B-REP}(T, A)$ such that $B_1(c) \in A'$ and $B_2(c) \notin A'$ that is obtained by just removing $B_2(c)$ from A and keep $B_1(c)$ (otherwise we will not have all maximal T -consistent subsets of A in $\text{B-REP}(T, A)$, which contradicts the definition of $\text{B-REP}(T, A)$ itself). Hence, we have a contradiction. Thus, there exists $A' \in \text{B-REP}(T, A)$ such that (i) $B_1(c) \in A'$, (ii) $B_2(c) \notin A'$. The proof for the case of negative role inclusion is similar. □

Similarly for the case of functionality assertion, below we show that for each role assertion that violates a functional assertion, there exists an ABox in the set of b-repair results that contains only this role assertion but not the other role assertions that together they violate the corresponding functional assertion.

Lemma 5.44. *Given a TBox T , and an ABox A , for every functional assertion ($\text{funct } R$), if $\{R(c, c_1), R(c, c_2), \dots, R(c, c_n)\} \subseteq A$ (for any constants $\{c, c_1, c_2, \dots, c_n\} \subseteq \Delta$), then there exist $A' \in \text{B-REP}(T, A)$ such that (i) $R(c, c_1) \in A'$, (ii) $R(c, c_2) \notin A', \dots, R(c, c_n) \notin A'$,*

Proof. Similar to the proof of Lemma 5.43. \square

Below, we show that the result of c-repair does not contain any ABox assertions that, together with another ABox assertions, violate a negative inclusion assertion. Intuitively, this fact is obtained by using Lemma 5.43 which said that for every pair of ABox assertions that violates some negative inclusion assertions, each of them will be contained in two different ABoxes in the results of b-repair. As a consequence, we have that both of them are not in the result of c-repair when we compute the intersection of all of b-repair results.

Lemma 5.45. *Given a TBox T , and an ABox A , for every negative concept inclusion assertion $B_1 \sqsubseteq \neg B_2$ such that $T \models B_1 \sqsubseteq \neg B_2$ and $B_1 \neq B_2$, if there exists $B_1(c) \in A$ (for any constant $c \in \Delta$) such that $B_1(c)$ violates $B_1 \sqsubseteq \neg B_2$, then $B_1(c) \notin \text{C-REP}(T, A)$. (Similarly for the case of negative role inclusion assertion).*

Proof. Let $\{B_1(c), B_2(c)\} \subseteq A$ (for any constant $c \in \Delta$), then by Lemma 5.43, there exist $A' \in \text{B-REP}(T, A)$ and $A'' \in \text{B-REP}(T, A)$ such that (i) $B_1(c) \in A'$, (ii) $B_2(c) \notin A'$, (iii) $B_2(c) \in A''$, and (iv) $B_1(c) \notin A''$. By Definition 5.2, $\text{C-REP}(T, A) = \bigcap_{A_i \in \text{B-REP}(T, A)} A_i$. Since $\{B_1(c), B_2(c)\} \not\subseteq A' \cap A''$, then we have that $\{B_1(c), B_2(c)\} \not\subseteq \text{C-REP}(T, A)$. Thus $B_1(c) \notin \text{C-REP}(T, A)$. The proof for the case of negative role inclusion is similar. \square

Similarly, below we show that the result of c-repair does not contain any role assertions that, together with another role assertions, violate a functional assertion. The intuition of the proof is similar to the proof of Lemma 5.45. I.e., they are thrown away when we compute the intersection of all of b-repair results.

Lemma 5.46. *Given a TBox T , and an ABox A , for every functionality assertion ($\text{funct } R$) $\in T$, if there exists $R(c, c_1) \in A$ (for some constants $\{c, c_1\} \subseteq \Delta$) such that $R(c, c_1)$ violates ($\text{funct } R$), then $R(c, c_1) \notin A'$.*

Proof. Similar to the proof of Lemma 5.45. \square

Now, in the two following Lemmas we show a property of a c-repair action, namely that a c-repair action deletes all ABox assertions that, together with another ABox assertions, violate a negative inclusion or a functionality assertion.

Lemma 5.47. *Given a TBox T , and an ABox A , a service call map m and a c-repair action α_c^T . If $(\langle A, m \rangle, \alpha_c^T \sigma, \langle A', m \rangle) \in \text{TELL}_{f_s}$ (where σ is an empty substitution), then for every negative concept inclusion assertion $B_1 \sqsubseteq \neg B_2$ such that $T \models B_1 \sqsubseteq \neg B_2$ and $B_1 \neq B_2$, if there exists $B_1(c) \in A$ (for any constant $c \in \Delta$) such that $B_1(c)$ violates*

$B_1 \sqsubseteq \neg B_2$, then $B_1(c) \notin A'$. (Similarly for the case of the negative role inclusion assertion).

Proof. Since $T \models B_1 \sqsubseteq \neg B_2$, by the definition of α_c^T , we have

$$q_{\text{unsat}}^n(B_1 \sqsubseteq \neg B_2, x) \rightsquigarrow \{\text{del } \{B_1(x), B_2(x)\}\} \in \text{EFF}(\alpha_c^T).$$

Since, $(\langle A, m \rangle, \alpha_c^T \sigma, \langle A', m \rangle) \in \text{TELL}_{f_S}$, by the definition of TELL_{f_S} , we have $\langle A, \text{ADD}(T, A, \alpha_c^T \sigma) \theta, \text{DEL}(T, A, \alpha_c^T \sigma), A' \rangle \in f_S$. Now, it is easy to see that by the definition of filter f_S and $\text{DEL}(T, A, \alpha_c^T \sigma)$, we have $B_1(c) \notin A'$. \square

Lemma 5.48. *Given a TBox T , an ABox A , a service call map m and a c-repair action α_c^T . If $(\langle A, m \rangle, \alpha_c^T \sigma, \langle A', m \rangle) \in \text{TELL}_{f_S}$ (where σ is an empty substitution) then for every functional assertion $(\text{funct } R) \in T$, if there exists $R(c, c_1) \in A$ (for some constants $\{c, c_1\} \subseteq \Delta$) such that $R(c, c_1)$ violates $(\text{funct } R)$, then $R(c, c_1) \notin A'$.*

Proof. Similar to the proof of Lemma 5.47. \square

Next, we show that every Abox assertion that does not violate any TBox assertions will appear in all results of a b-repair.

Lemma 5.49. *Given a TBox T , and an ABox A , for every concept assertion $C(c) \in A$ (for any constant $c \in \Delta$) such that $C(c) \notin \text{INC}(A)$, it holds that for every $A' \in \text{B-REP}(T, A)$, we have $C(c) \in A'$. (Similarly for role assertion).*

Proof. Suppose by contradiction there exists $C(c) \in A$ such that $C(c) \notin \text{INC}(A)$ and there exists $A' \in \text{B-REP}(T, A)$, such that $C(c) \notin A'$. Then, since $C(c) \notin \text{INC}(A)$, there exists A'' such that $A' \subset A'' \subseteq A$ and A'' is T -consistent (where $A'' = A' \cup \{C(c)\}$). Hence, $A' \notin \text{B-REP}(T, A)$. Thus we have a contradiction. Therefore the claim is proven. The proof for the case of role assertion can be done similarly. \square

From the previous Lemma, we can show that every ABox assertion that does not violate any TBox assertion will appear in the c-repair results.

Lemma 5.50. *Given a TBox T , and an ABox A , for every concept assertion $C(c) \in A$ s.t. $C(c) \notin \text{INC}(A)$ we have that $C(c) \in \text{C-REP}(T, A)$. (Similarly for role assertion).*

Proof. Let $C(c) \in A$ be any arbitrary concept assertion s.t. $C(c) \notin \text{INC}(A)$. By Lemma 5.49, for every $A' \in \text{B-REP}(T, A)$, we have $C(c) \in A'$. Hence, since $\text{C-REP}(T, A) = \bigcap_{A_i \in \text{B-REP}(T, A)} A_i$, we have $C(c) \in \text{C-REP}(T, A)$. The proof for the case of role assertion can be done similarly. \square

Finally, below we can show that the c-repair action is correctly mimicked the c-repair computation. I.e., they produce the same results.

Theorem 5.51. *Given a TBox T , an ABox A , a service call map m and a c-repair action α_c^T . Let $A_1 = \text{C-REP}(T, A)$, and $\langle \langle A, m \rangle, \alpha_c^T \sigma, \langle A_2, m \rangle \rangle \in \text{TELL}_{f_S}$ where σ is an empty substitution, then we have $A_1 = A_2$.*

Proof. The core idea of the proof is as follows: since c-repair take the intersection of all b-repairs, and the behavior of b-repair is to choose one assertion among those conflicting assertions, we have that basically c-repair throw away all assertions that involve in making inconsistency. As it can be seen from the construction of c-repair action α_c^T , such an action basically also throw away all assertions that involve in making inconsistency. Moreover, both c-repair and c-repair action do nothing regarding those ABox assertions that do not involve in any inconsistency. Technically, the claim is proven by the fact that α_c^T never deletes any ABox assertion that does not involve in any source of inconsistency, and also by using Lemmas 5.45 to 5.48 and 5.50. \square

5.3.2.3 Reducing the Verification of C-GKABs Into S-GKABs

In the following two Lemmas, we aim to show that the transition systems of a C-GKAB and its corresponding S-GKAB (obtained through τ_C) are S-bisimilar.

Lemma 5.52. *Let \mathcal{G} be a C-GKAB with transition system $\Upsilon_{\mathcal{G}}^{fC}$, and let $\tau_C(\mathcal{G})$ be an S-GKAB with transition system $\Upsilon_{\tau_C(\mathcal{G})}^{fs}$ obtained through τ_C . Consider a state $\langle A_c, m_c, \delta_c \rangle$ of $\Upsilon_{\mathcal{G}}^{fC}$ and a state $\langle A_s, m_s, \delta_s \rangle$ of $\Upsilon_{\tau_C(\mathcal{G})}^{fs}$. If $A_s = A_c$, $m_s = m_c$ and $\delta_s = \kappa_C(\delta_c)$, then $\langle A_c, m_c, \delta_c \rangle \sim_{\text{so}} \langle A_s, m_s, \delta_s \rangle$.*

Proof. Let

1. $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$, and $\Upsilon_{\mathcal{G}}^{fC} = \langle \Delta, T, \Sigma_c, s_{0c}, abox_c, \Rightarrow_c \rangle$,
2. $\tau_C(\mathcal{G}) = \langle T_s, A_0, \Gamma_s, \delta_s \rangle$, and $\Upsilon_{\tau_C(\mathcal{G})}^{fs} = \langle \Delta, T_s, \Sigma_s, s_{0s}, abox_s, \Rightarrow_s \rangle$.

Now, we have to show the following: For every state $\langle A_c'', m_c'', \delta_c'' \rangle$ such that

$$\langle A_c, m_c, \delta_c \rangle \Rightarrow \langle A_c'', m_c'', \delta_c'' \rangle,$$

there exists states $\langle A_s', m_s', \delta_s' \rangle$ and $\langle A_s'', m_s'', \delta_s'' \rangle$ such that:

- (a) we have $\langle A_s, m_s, \delta_s \rangle \Rightarrow_s \langle A_s', m_s', \delta_s' \rangle \Rightarrow_s \langle A_s'', m_s'', \delta_s'' \rangle$
- (b) $A_s'' = A_c''$;
- (c) $m_s'' = m_c''$;
- (d) $\delta_s'' = \kappa_C(\delta_c'')$.

By definition of $\Upsilon_{\mathcal{G}}^{fC}$, since $\langle A_c, m_c, \delta_c \rangle \Rightarrow \langle A_c'', m_c'', \delta_c'' \rangle$, we have $\langle A_c, m_c, \delta_c \rangle \xrightarrow{\alpha\sigma_c, fC} \langle A_c'', m_c'', \delta_c'' \rangle$. Furthermore, by the definition of $\xrightarrow{\alpha\sigma_c, fC}$, we have that:

- $\langle \langle A_c, m_c \rangle, \alpha\sigma_c, \langle A_c'', m_c'' \rangle \rangle \in \text{TELL}_{fC}$, and
- σ_c is a legal parameter assignment for α in A_c w.r.t. **pick** $Q(\vec{p}).\alpha(\vec{p})$ (i.e., $\text{ASK}(Q\sigma_c, T, A_c) = \text{true}$).

Since $\langle \langle A_c, m_c \rangle, \alpha\sigma_c, \langle A_c'', m_c'' \rangle \rangle \in \text{TELL}_{fC}$, by the definition of TELL_{fC} , there exists $\theta_c \in \text{EVAL}(\text{ADD}(T, A_c, \alpha\sigma_c))$ such that

- θ_c and m_c agree on the common values in their domains.
- $m_c'' = m_c \cup \theta_c$.
- $(A_c, \text{ADD}(T, A_c, \alpha\sigma_c)\theta_c, \text{DEL}(T, A_c, \alpha\sigma_c), A_c'') \in f_C$.
- A_c'' is T -consistent.

Since $\langle A_c, \text{ADD}(T, A_c, \alpha\sigma_c)\theta_c, \text{DEL}(T, A_c, \alpha\sigma_c), A_c'' \rangle \in f_C$, by the definition of f_C , there exists A_c' such that $A_c'' \in \text{C-REP}(T, A_c')$ where $A_c' = (A_c \setminus \text{DEL}(T, A_c, \alpha\sigma_c)) \cup \text{ADD}(T, A_c, \alpha\sigma_c)\theta_c$. Furthermore, since $\delta_s = \kappa_C(\delta_c)$, by the definition of κ_C , we have that

$$\kappa_C(\text{pick } Q(\vec{p}).\alpha(\vec{p})) = \text{pick } Q(\vec{p}).\alpha'(\vec{p}); \text{pick true}.\alpha_c^T().$$

Hence, we have that the next executable part of program on state $\langle A_s, m_s, \delta_s \rangle$ is

$$\mathbf{pick} \ Q(\vec{p}).\alpha'(\vec{p}); \mathbf{pick} \ \mathbf{true}.\alpha_c^T().$$

Now, since σ_c maps parameters of $\alpha \in \Gamma$ to constants in $\text{ADOM}(A_c)$, and $A_c = A_s$, we can construct σ_s mapping parameters of $\alpha' \in \Gamma_s$ to constants in $\text{ADOM}(A_s)$ such that $\sigma_c = \sigma_s$. Moreover, since $A_s = A_c$, the certain answers computed over A_c are the same to those computed over A_s . Hence, $\alpha' \in \Gamma_s$ is executable in A_s with (legal parameter assignment) σ_s . Now, since we have $m_s = m_c$, then we can construct θ_s such that $\theta_s = \theta_c$. Hence, we have the following:

- θ_s and m_s agree on the common values in their domains.
- $m'_s = \theta_s \cup m_s = \theta_c \cup m_c = m''_c$.

Let $A'_s = (A_s \setminus \text{DEL}(T_s, A_s, \alpha'\sigma_s)) \cup \text{ADD}(T_s, A_s, \alpha'\sigma_s)\theta_s$, as a consequence, we have $\langle A_s, \text{ADD}(T_s, A_s, \alpha'\sigma_s)\theta_s, \text{DEL}(T_s, A_s, \alpha'\sigma_s), A'_s \rangle \in f_S$. Since $A_s = A_c$, $\theta_s = \theta_c$, and $\sigma_s = \sigma_c$, it follows that

- $\text{DEL}(T_s, A_s, \alpha'\sigma_s) = \text{DEL}(T, A_c, \alpha\sigma_c)$, and
- $\text{ADD}(T_s, A_s, \alpha'\sigma_s)\theta_s \setminus \{\text{State}(\text{temp})\} = \text{ADD}(T, A_c, \alpha\sigma_c)\theta_c$.

Hence, by the construction of A'_s and A'_c above, we also have $A'_s \setminus \{\text{State}(\text{temp})\} = A'_c$. By the definition of τ_C , we have $T_s = T_p$ (i.e., only positive inclusion assertion of T), hence A'_s is T_s -consistent. Thus, by the definition of TELL_{f_s} , we have $\langle \langle A_s, m_s \rangle, \alpha'\sigma_s, \langle A'_s, m'_s \rangle \rangle \in \text{TELL}_{f_s}$. Moreover, we have

$$\langle A_s, m_s, \mathbf{pick} \ Q(\vec{p}).\alpha'(\vec{p}); \delta_0 \rangle \xrightarrow{\alpha'\sigma_s, f_s} \langle A'_s, m'_s, \delta_0 \rangle$$

where $\delta_0 = \mathbf{pick} \ \mathbf{true}.\alpha_c^T()$. Now, it is easy to see that

$$\langle A'_s, m'_s, \mathbf{pick} \ \mathbf{true}.\alpha_c^T() \rangle \xrightarrow{\alpha_c^T\sigma, f_s} \langle A''_s, m''_s, \varepsilon \rangle$$

where

- $m''_s = m'_s$ (since α_c^T does not involve any service call),
- σ is empty substitution (because α_c^T is a 0-ary action),
- $\langle A''_s, m''_s, \varepsilon \rangle$ is a final state,
- $\text{State}(\text{temp}) \notin A''_s$,
- $A''_s = \text{C-REP}(T, A'_s \setminus \{\text{State}(\text{temp})\})$ (by Theorem 5.51, and by considering that $\text{State}(\text{temp})$ is only a special marker).

Since $A'_s \setminus \{\text{State}(\text{temp})\} = A'_c$, $A''_s = \text{C-REP}(T, A'_s \setminus \{\text{State}(\text{temp})\})$, and $A''_c = \text{C-REP}(T, A'_c)$, then it is easy to see that $A''_s = A''_c$. Moreover, since $\langle A''_s, m''_s, \varepsilon \rangle$ is a final state, we have successfully finished executing

$$\mathbf{pick} \ Q(\vec{p}).\alpha'(\vec{p}); \mathbf{pick} \ \mathbf{true}.\alpha_c^T(),$$

and by the definition of κ_C the rest of the program to be executed is $\delta''_s = \kappa_C(\delta''_c)$. Thus, we have

$$\langle A_s, m_s, \delta_s \rangle \Rightarrow_s \langle A'_s, m'_s, \delta'_s \rangle \Rightarrow_s \langle A''_s, m''_s, \delta''_s \rangle$$

where

- (a) $A''_s = A''_c$;
- (b) $m''_s = m''_c$;
- (c) $\delta''_s = \kappa_C(\delta''_c)$.

The other direction of bisimulation relation can be proven in a similar way. \square

Having Lemma 5.52 in hand, we can easily show that given a C-GKAB, its transition system is S-bisimilar to the transition of its corresponding S-GKAB that is obtained via the translation τ_C as follows.

Lemma 5.53. *Given a C-GKAB \mathcal{G} , we have $\Upsilon_{\mathcal{G}}^{fc} \sim_{\text{so}} \Upsilon_{\tau_C(\mathcal{G})}^{fs}$*

Proof. Let

1. $\mathcal{G} = \langle T, A_0, \Gamma, \delta_c \rangle$, and $\Upsilon_{\mathcal{G}}^{fc} = \langle \Delta, T, \Sigma_c, s_{0c}, \text{abox}_c, \Rightarrow_c \rangle$,
2. $\tau_C(\mathcal{G}) = \langle T_s, A_0, \Gamma_s, \delta_s \rangle$, and $\Upsilon_{\tau_C(\mathcal{G})}^{fs} = \langle \Delta, T_s, \Sigma_s, s_{0s}, \text{abox}_s, \Rightarrow_s \rangle$.

We have that $s_{0c} = \langle A_0, m_c, \delta_c \rangle$ and $s_{0s} = \langle A_0, m_s, \delta_s \rangle$ where $m_c = m_s = \emptyset$. By the definition of κ_C and τ_C , we also have $\delta_s = \kappa_C(\delta_c)$. Hence, by Lemma 5.52, we have $s_{0c} \sim_{\text{so}} s_{0s}$. Therefore, by the definition of S-bisimulation, we have $\Upsilon_{\mathcal{G}}^{fc} \sim_{\text{so}} \Upsilon_{\tau_C(\mathcal{G})}^{fs}$. \square

Finally, we are now ready to show that the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ formulas over C-GKABs can be recast as verification of $\mu\mathcal{L}_A^{\text{EQL}}$ formulas over S-GKAB as follows.

Theorem 5.54. *Given a C-GKAB \mathcal{G} and a closed $\mu\mathcal{L}_A^{\text{EQL}}$ property Φ ,*

$$\Upsilon_{\mathcal{G}}^{fc} \models \Phi \text{ iff } \Upsilon_{\tau_C(\mathcal{G})}^{fs} \models t_{\text{dup}}(\Phi)$$

Proof. By Lemma 5.53, we have that $\Upsilon_{\mathcal{G}}^{fc} \sim_{\text{so}} \Upsilon_{\tau_C(\mathcal{G})}^{fs}$. Hence, by Lemma 5.42, it is easy to see that the claim is proved. \square

5.3.3 From E-GKABs to Standard GKABs

Differently from the case of B-GKABs and C-GKABs, the case of E-GKABs pose two challenges while translating it into S-GKABs:

1. when applying an atomic action (and managing the possibly arising inconsistency) it is necessary to distinguish those assertions that are newly introduced by the action from those already present in the system;
2. the evolution semantics can be applied only if the assertions to be added are consistent with the TBox, and hence an additional check is required to abort the action execution if this is not the case.

To this aim, given a TBox T , we duplicate concepts and roles in T , introducing a fresh concept name N^n for every concept name N in T (similarly for roles). The key idea is to insert those constants that are added to N also in N^n , so as to trace that they are part of the update.

The first issue described above is then tackled by compiling the bold evolution semantics into a 0-ary *evolution action* α_e^T as follows:

Evolution Action

Definition 5.55 (Evolution Action). Given a TBox T , we define an *evolution action* α_e^T over T as a 0-ary (i.e., has no action parameters), where $\text{EFF}(\alpha_e^T)$ is the smallest set of effects containing:

- for each assertion $(\text{funct } R) \in T$, we have
$$\exists z. q_{\text{unsat}}^f((\text{funct } R), x, y, z) \wedge R^n(x, y) \rightsquigarrow \text{del } \{R(x, z)\} \in \text{EFF}(\alpha_e^T),$$

- for each negative concept inclusion assertion $B_1 \sqsubseteq \neg B_2$ such that $T \models B_1 \sqsubseteq \neg B_2$, we have

$$q_{\text{unsat}}^n(B_1 \sqsubseteq \neg B_2, x) \wedge B_1^n(x) \rightsquigarrow \mathbf{del} \{B_2(x)\} \in \text{EFF}(\alpha_e^T),^3$$
- for each negative role inclusion assertion $R_1 \sqsubseteq \neg R_2$ such that $T \models R_1 \sqsubseteq \neg R_2$, we have:

$$q_{\text{unsat}}^n(R_1 \sqsubseteq \neg R_2, x, y) \wedge R_1^n(x, y) \rightsquigarrow \mathbf{del} \{R_2(x, y)\} \in \text{EFF}(\alpha_e^T),$$
- for each concept name $N \in \text{VOC}(T)$, we have:

$$N^n(x) \rightsquigarrow \mathbf{del} \{N^n(x)\} \in \text{EFF}(\alpha_e^T),$$
- for each role name $P \in \text{VOC}(T)$, we have:

$$P^n(x, y) \rightsquigarrow \mathbf{del} \{P^n(x, y)\} \in \text{EFF}(\alpha_e^T).$$
- $\mathbf{true} \rightsquigarrow \mathbf{del} \{\text{State}(\text{temp})\} \in \text{EFF}(\alpha_e^T).$

■

Those effects in $\text{EFF}(\alpha_e^T)$ mirror those of Section 5.3.2, with the difference that they asymmetrically remove old assertions when inconsistency arises. The last two bullets guarantee that the content of concept and role names tracking the newly added assertions are flushed.

We now define a translation function κ_E that essentially concatenates each action invocation with an evolution action in order to simulate the action executions in E-GKABs. Additionally, the translation function κ_E also serves as a one-to-one correspondence (bijection) between the original and the translated program (as well as between the sub-program).

Definition 5.56 (Program Translation κ_E). Given an E-GKAB $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$, we define a *translation* κ_E that translates a program δ into a program δ' inductively as follows: Program Translation
 κ_E

$$\begin{aligned}
\kappa_E(\mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p})) &= \mathbf{pick} \ Q(\vec{p}).\alpha'(\vec{p}); \mathbf{pick} \ \mathbf{true}.\alpha_e^T() \\
\kappa_E(\varepsilon) &= \varepsilon \\
\kappa_E(\delta_1 | \delta_2) &= \kappa_E(\delta_1) | \kappa_E(\delta_2) \\
\kappa_E(\delta_1; \delta_2) &= \kappa_E(\delta_1); \kappa_E(\delta_2) \\
\kappa_E(\mathbf{if} \ \varphi \ \mathbf{then} \ \delta_1 \ \mathbf{else} \ \delta_2) &= \mathbf{if} \ \varphi \ \mathbf{then} \ \kappa_E(\delta_1) \ \mathbf{else} \ \kappa_E(\delta_2) \\
\kappa_E(\mathbf{while} \ \varphi \ \mathbf{do} \ \delta) &= \mathbf{while} \ \varphi \ \mathbf{do} \ \kappa_E(\delta)
\end{aligned}$$

where

- action $\alpha'(\vec{p})$ is obtained from $\alpha(\vec{p}) \in \Gamma$, such that for each effect

$$[q^+] \wedge Q^- \rightsquigarrow \mathbf{add} \ F^+, \mathbf{del} \ F^- \in \text{EFF}(\alpha)$$

we have

$$[q^+] \wedge Q^- \rightsquigarrow \mathbf{add} \ F^+ \cup F^{+n}, \mathbf{del} \ F^- \in \text{EFF}(\alpha')$$

where F^{+n} duplicates F^+ by using the vocabulary for newly introduced atoms and additionally, we also have

$$\mathbf{true} \rightsquigarrow \mathbf{add} \ \{\text{State}(\text{temp})\} \in \text{EFF}(\alpha')$$

³ Note: if $B_1 = \exists P_1$ (resp. $B_2 = \exists P_2$ and $B_1^n = \exists P_1^n$), then with a little abuse of notation, specifically for this case, we have that the atom $B_1(x)$ (resp. $B_2(x)$ and $B_1^n(x)$) denotes $P_1(x, y)$ (resp. $P_2(x, y)$ and $P_1^n(x, y)$). For instance, for the assertion $\exists P_1 \sqsubseteq \neg N$, we have $P_1(x, y) \wedge N(x) \wedge P_1^n(x, y) \rightsquigarrow \mathbf{del} \{N(x)\} \in \text{EFF}(\alpha_e^T)$. Similarly when $B_1 = \exists P_1^-$ (resp. $B_2 = \exists P_2^-$ and $B_1^n = \exists P_1^{n-}$).

- α_e^T is an evolution action over T .

■

We then define a translation τ_E that transforms an E-GKAB to an S-GKAB as follows.

*Translation from
E-GKAB to S-GKAB*

Definition 5.57 (Translation from E-GKAB to S-GKAB). We define a translation τ_E that, given an E-GKAB $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$, generates an S-GKAB $\tau_E(\mathcal{G}) = \langle T_p \cup T^n, A_0, \Gamma' \cup \{\alpha_e^T\}, \delta' \rangle$, where:

- T_p is the positive inclusion assertions of T (see Definition 2.12),
- T^n is obtained from T by renaming each concept name N in T into N^n (similarly for roles). In this way, the original concepts/roles are only subject in $\tau_E(\mathcal{G})$ to the positive inclusion assertions of T (i.e., T_p), while concepts/roles tracking newly inserted assertions are subject also to negative constraints. This blocks the generation of the successor state when the assertions to be added to the current ABox are T -inconsistent.
- Γ' is obtained by translating each action in $\alpha(\vec{p}) \in \Gamma$ into action $\alpha'(\vec{p})$, such that for each effect

$$[q^+] \wedge Q^- \rightsquigarrow \mathbf{add} F^+, \mathbf{del} F^- \in \text{EFF}(\alpha)$$

we have

$$[q^+] \wedge Q^- \rightsquigarrow \mathbf{add} F^+ \cup F^{+n}, \mathbf{del} F^- \in \text{EFF}(\alpha')$$

where F^{+n} duplicates F^+ by using the vocabulary for newly introduced atoms and additionally, we also have

$$\text{true} \rightsquigarrow \mathbf{add} \{\text{State}(\text{temp})\} \in \text{EFF}(\alpha')$$

- $\delta' = \kappa_E(\delta)$.

■

By exploiting the same $\mu\mathcal{L}_A^{\text{EQL}}$ translation used in Section 5.3.2 (i.e., the translation t_{dup} in Definition 5.39), we will show later that $\gamma_{\mathcal{G}}^{f_E} \models \Phi$ if and only if $\gamma_{\tau_E(\mathcal{G})}^{f_S} \models t_{\text{dup}}(\Phi)$. Hence reducing the $\mu\mathcal{L}_A^{\text{EQL}}$ verification over E-GKABs to S-GKABs. The strategy of the proof is similar to the reduction from the verification of C-GKABs into the verification of S-GKABs in Section 5.3.2. I.e., to show that the transition system of an E-GKAB is S-bisimilar to the transition system of its corresponding S-GKAB, and hence they can not be distinguish by any $\mu\mathcal{L}_A^{\text{EQL}}$ formulas modulo translation t_{dup} .

5.3.3.1 Recasting the Verification of E-GKABs Into S-GKABs

As the first step, we show an important property of the filter f_E (which is also a property of EVOL operator). Specifically, we show that every ABox assertion in the evolution result is either a new assertion or it was already in the original ABox and it was not deleted as well as did not violate any TBox constraints (together with another ABox assertions). Formally the claim is stated below.

Lemma 5.58. *Given a TBox T , a T -consistent ABox A , a T -consistent set F^+ of ABox assertions to be added, and a set F^- of ABox assertions to be deleted such that $A_e = \text{EVOL}(T, A, F^+, F^-)$, we have $N(c) \in A_e$ if and only if either*

1. $N(c) \in F^+$, or
2. $N(c) \in (A \setminus F^-)$ and there does not exist $B(c) \in F^+$ such that $T \models N \sqsubseteq \neg B$.
(Similarly for the case of role assertion with the corresponding violation of negative role inclusion or functional assertion).

Proof. The intuition of the correctness of this claim is simply obtained from the definition of bold-evolution operator itself (cf. Definition 5.4). I.e.,

$$\text{EVOL}(T, A, F^+, F^-) = F^+ \cup A'$$

where

- F^+ is a set of newly added assertions,
- A' is a set of ABox assertions that is obtained from A by throwing away those assertions that are either also belongs to F^- or have a conflict with some assertions in F^+ .

Hence, it is easy to see that an assertion is belong to the result of the application of bold-evolution if it is either

1. a newly added assertion, or
2. an assertion that is not deleted and doesn't have a conflict with the newly added assertion.

Technically, the proof is as follows:

“ \implies ”: Assume $N(c) \in A_e$, since $A_e = \text{EVOL}(T, A, F^+, F^-)$, by the definition of $\text{EVOL}(T, A, F^+, F^-)$, we have $A_e = F^+ \cup A'$, where

1. $A' \subseteq (A \setminus F^-)$,
2. $F^+ \cup A'$ is T -consistent, and
3. there does not exist A'' such that $A' \subset A'' \subseteq (A \setminus F^-)$ and $F^+ \cup A''$ is T -consistent.

Hence, we have either

- (1) $N(c) \in F^+$, or
- (2) $N(c) \in A'$.

For the case (2), as a consequence:

- Since $N(c) \in A'$ and $A' \subseteq (A \setminus F^-)$ it follows that $N(c) \in (A \setminus F^-)$.
- Since $F^+ \cup A'$ is T -consistent, then we have that there does not exist $B(c) \in F^+$ s.t. $T \models N \sqsubseteq \neg B$.

Thus, the claim is proven.

“ \impliedby ”: We divide the proof into two parts:

- (1) Assume $N(c) \in F^+$. Then simply by the definition of $\text{EVOL}(T, A, F^+, F^-)$, we have $N(c) \in A_e$.
- (2) Supposed by contradiction we have that $N(c) \in (A \setminus F^-)$ and there does not exist $B(c) \in F^+$ s.t. $T \models N \sqsubseteq \neg B$, and $N(c) \notin A_e$. Since $N(c) \notin A_e$, by the definition of $\text{EVOL}(T, A, F^+, F^-)$, we have that $N(c) \notin F^+$ and $N(c) \notin A'$ in which A' should satisfies the following:
 - $A' \subseteq (A \setminus F^-)$,
 - $F^+ \cup A'$ is T -consistent, and
 - there does not exist A'' such that $A' \subset A'' \subseteq (A \setminus F^-)$ and $F^+ \cup A''$ is T -consistent.

But then we have a contradiction since there exists $A'' = A' \cup \{N(c)\}$ such that $A' \subset A'' \subseteq (A \setminus F^-)$ and $F^+ \cup A''$ is T -consistent. Hence, we must have $N(c) \in A_e$. \square

Now we show a property of evolution action α_e^T which says that every ABox assertion in the result of the execution of α_e^T is either a newly added assertion, or an old assertion that does not violate any TBox constraints. Precisely we state this property below.

Lemma 5.59. *Given*

- an E-GKAB $\mathcal{G} = \langle T, A_0, \Gamma_e, \delta_e \rangle$ with transition system $\mathcal{R}_{\mathcal{G}}^{fE}$, and
- an S-GKAB $\tau_E(\mathcal{G}) = \langle T_s, A_0, \Gamma_s, \delta_s \rangle$ (with transition system $\mathcal{R}_{\tau_E(\mathcal{G})}^{fS}$) that is obtained from \mathcal{G} through τ_E , where $T_s = T_p \cup T^n$.

Let $\langle A, m, \delta \rangle$ be any state in $\mathcal{R}_{\tau_E(\mathcal{G})}^{fS}$, $\alpha' \in \Gamma_s$ be any action, A is T_s -consistent and does not contain any ABox assertions constructed from $\text{VOC}(T^n)$ and we have:

$$\langle A, m, \delta \rangle \xrightarrow{\alpha'\sigma, fS} \langle A', m', \delta' \rangle \xrightarrow{\alpha_e^T \sigma', fS} \langle A'', m'', \delta'' \rangle$$

for

- a particular legal parameter assignment σ
- an empty substitution σ' ,
- a particular service call evaluation $\theta \in \text{EVAL}(\text{ADD}(T, A, \alpha'\sigma))$ that agree with m on the common values in their domains.

We have $N(c) \in A''$ if and only if N is not in the vocabulary of TBox T^n and either

1. $N(c) \in \text{ADD}(T_s, A, \alpha'\sigma)\theta$, or
2. $N(c) \in (A \setminus \text{DEL}(T_s, A, \alpha'\sigma))$ and there does not exists $B(c) \in \text{ADD}(T_s, A, \alpha'\sigma)\theta$ such that $T \models N \sqsubseteq \neg B$.

(Similarly for the case of role assertion with the corresponding violation of negative role inclusion or functional assertion).

Proof. First of all, it will not be the case that $N(c)$ is **State(temp)** since in any case α_e^T deletes **State(temp)**. The proof of this Lemma is then divided into two parts as follows:

“ \implies ”: Assume $N(c) \in A''$, since the evolution action α_e^T only

1. removes old assertions when inconsistency arises,
2. flushes every ABox assertions constructed by the vocabulary of T^n ,

then we have the following:

1. N is not in the vocabulary of TBox T^n (otherwise it will be flushed by α_e^T)
2. $N(c) \in A'$ (because α_e^T never introduce a new ABox assertion),
3. if there exists $B(c) \in A'$ such that $T \models N \sqsubseteq \neg B$, then $B(c) \notin A''$, $B^n(c) \notin A'$, and $N^n(c) \in A'$ (i.e., if $N(c) \in A'$ violates a negative inclusion assertion, $N(c)$ must be a newly added ABox assertion, otherwise it will be deleted by α_e^T).

Now, since A and A' are T_s -consistent (because $\langle A, m, \delta \rangle \xrightarrow{\alpha'\sigma, fS} \langle A', m', \delta' \rangle$), then $\text{ADD}(T, A, \alpha'\sigma)\theta$ is T_s -consistent. Hence we have either

1. $N(c) \in \text{ADD}(T, A, \alpha'\sigma)\theta$ (and there does not exists $B(c)$ such that $B(c) \in \text{ADD}(T, A, \alpha'\sigma)\theta$, and $T \models N \sqsubseteq \neg B$), or

2. $N(c) \in (A \setminus \text{DEL}(T, A, \alpha'\sigma))$ and there does not exists $B(c) \in \text{ADD}(T, A, \alpha'\sigma)\theta$ such that $T \models N \sqsubseteq \neg B$ (otherwise we have $\{N(c), B(c), B^n(c)\} \subseteq A'$ and then $N(c)$ will be deleted by α_e^T).

Therefore, the claim is proved.

“ \Leftarrow ”: Assume N is not in the vocabulary of $\text{TBox } T^n$, then we divide the proof into two parts:

1. Assume $N(c) \in \text{ADD}(T, A, \alpha'\sigma)\theta$. Then, by the construction of α' and the definition of $\frac{\alpha'\sigma, f_S}{\rightarrow}$, it is easy to see that $\{N(c), N^n(c)\} \subseteq A'$. Therefore $N(c) \in A''$ (by construction of α_e^T).
2. Assume $N(c) \in (A \setminus \text{DEL}(T, A, \alpha'\sigma))$ and there does not exists $B(c) \in \text{ADD}(T, A, \alpha'\sigma)\theta$ such that $T \models N \sqsubseteq \neg B$. Hence, by the definition of $\frac{\alpha'\sigma, f_S}{\rightarrow}$, we have $N(c) \in A'$. Moreover, because $N(c) \in A'$ does not violate any negative inclusion assertions, by construction of α_e^T , we also simply have $N(c) \in A''$.

□

Next, in the following two Lemmas we aim to show that the transition system of an E-GKAB is S-bisimilar to the transition system of its corresponding S-GKAB that is obtained from translation τ_E .

Lemma 5.60. *Let $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ be an E-GKAB with transition system $\Upsilon_{\mathcal{G}}^{f_E}$, and let $\tau_E(\mathcal{G}) = \langle T_s, A_0, \Gamma_s, \delta_s \rangle$ be an S-GKAB with transition system $\Upsilon_{\tau_E(\mathcal{G})}^{f_S}$ obtained through τ_E . Consider a state $\langle A_e, m_e, \delta_e \rangle$ of $\Upsilon_{\mathcal{G}}^{f_E}$ and a state $\langle A_s, m_s, \delta_s \rangle$ of $\Upsilon_{\tau_E(\mathcal{G})}^{f_S}$. If $A_s = A_e$, $m_s = m_e$, A_s is T_s -consistent and $\delta_s = \kappa_E(\delta_e)$, then $\langle A_e, m_e, \delta_e \rangle \sim_{\text{so}} \langle A_s, m_s, \delta_s \rangle$.*

Proof. For the simplicity of the proof, here we ignore the presence of $\text{State}(\text{temp})$ that acts as a special marker (that marks an intermediate state). The important thing to observe is that $\text{State}(\text{temp})$ is always added to the intermediate state (where we need to execute the evolution action) but then it will be deleted after that. Now, let $\Upsilon_{\mathcal{G}}^{f_E} = \langle \Delta, T, \Sigma_e, s_{0e}, \text{abox}_e, \Rightarrow_e \rangle$, and $\Upsilon_{\tau_E(\mathcal{G})}^{f_S} = \langle \Delta, T_s, \Sigma_s, s_{0s}, \text{abox}_s, \Rightarrow_s \rangle$. We have to show the following: for every state $\langle A_e'', m_e'', \delta_e'' \rangle$ such that

$$\langle A_e, m_e, \delta_e \rangle \Rightarrow \langle A_e'', m_e'', \delta_e'' \rangle,$$

there exist states $\langle A_s', m_s', \delta_s' \rangle$ and $\langle A_s'', m_s'', \delta_s'' \rangle$ such that:

- (a) $\langle A_s, m_s, \delta_s \rangle \Rightarrow_s \langle A_s', m_s', \delta_s' \rangle \Rightarrow_s \langle A_s'', m_s'', \delta_s'' \rangle$
- (b) $A_s'' = A_e''$;
- (c) $m_s'' = m_e''$;
- (d) $\delta_s'' = \kappa_E(\delta_e'')$.

By definition of $\Upsilon_{\mathcal{G}}^{f_E}$, since $\langle A_e, m_e, \delta_e \rangle \Rightarrow \langle A_e'', m_e'', \delta_e'' \rangle$, we have $\langle A_e, m_e, \delta_e \rangle \xrightarrow{\alpha\sigma_e, f_E} \langle A_e'', m_e'', \delta_e'' \rangle$. Hence, by the definition of $\frac{\alpha\sigma_e, f_E}{\rightarrow}$, we have:

- $\langle \langle A_e, m_e \rangle, \alpha\sigma_e, \langle A_e'', m_e'' \rangle \rangle \in \text{TELL}_{f_E}$, and
- σ_e is a legal parameter assignment for α in A_e w.r.t. **pick** $Q(\vec{p}).\alpha(\vec{p})$ (i.e., $\text{ASK}(Q\sigma_e, T, A_e) = \text{true}$).

Since $\langle \langle A_e, m_e \rangle, \alpha\sigma_e, \langle A_e'', m_e'' \rangle \rangle \in \text{TELL}_{f_E}$, by the definition of TELL_{f_E} , there exists $\theta_e \in \text{EVAL}(\text{ADD}(T, A_e, \alpha\sigma_e))$ such that

- θ_e and m_e agree on the common values in their domains.
- $m_e'' = m_e \cup \theta_e$.
- $\langle A_e, \text{ADD}(T, A_e, \alpha\sigma_e)\theta_e, \text{DEL}(T, A_e, \alpha\sigma_e), A_e'' \rangle \in f_E$.
- A_e'' is T -consistent.

Since $\langle A_e, \text{ADD}(T, A_e, \alpha\sigma_e)\theta_e, \text{DEL}(T, A_e, \alpha\sigma_e), A_e'' \rangle \in f_E$, by the definition of f_E , we have

- $\text{ADD}(T, A_e, \alpha\sigma_e)\theta_e$ is T -consistent.
- $A_e'' = \text{EVOL}(T, A_e, \text{ADD}(T, A_e, \alpha\sigma_e)\theta_e, \text{DEL}(T, A_e, \alpha\sigma_e))$.

Furthermore, since $\delta_s = \kappa_E(\delta_e)$, by the definition of κ_E , we have that

$$\kappa_E(\mathbf{pick} Q(\vec{p}).\alpha'(\vec{p})) = \mathbf{pick} Q(\vec{p}).\alpha'(\vec{p}); \mathbf{pick} \text{true}.\alpha_e^T()$$

Hence, the part of program that we need to execute on state $\langle A_s, m_s, \delta_s \rangle$ is

$$\mathbf{pick} Q(\vec{p}).\alpha'(\vec{p}); \mathbf{pick} \text{true}.\alpha_e^T().$$

Now, since:

- $\alpha' \in \Gamma_s$ is obtained from $\alpha \in \Gamma$ through τ_E ,
- the translation τ_E transform α into α' without changing its parameters,
- σ_e maps parameters of $\alpha \in \Gamma$ to constants in $\text{ADOM}(A_e)$

then we can construct σ_s mapping parameters of $\alpha' \in \Gamma_s$ to constants in $\text{ADOM}(A_s)$ such that $\sigma_s = \sigma_e$. Moreover, since $A_s = A_e$, we know that the certain answers computed over A_e are the same to those computed over A_s . Hence $\alpha' \in \Gamma_s$ is executable in A_s with (legal parameter assignment) σ_s . Furthermore, since $m_s = m_e$, then we can construct θ_s , such that $\theta_s = \theta_e$. Hence, we have the following:

- θ_s and m_s agree on the common values in their domains.
- $m_s' = \theta_s \cup m_s = \theta_e \cup m_e = m_e''$.

Let $A_s' = (A_s \setminus \text{DEL}(T_s, A_s, \alpha'\sigma_s)) \cup \text{ADD}(T_s, A_s, \alpha'\sigma_s)\theta_s$, as a consequence, we have $\langle A_s, \text{ADD}(T_s, A_s, \alpha'\sigma_s)\theta_s, \text{DEL}(T_s, A_s, \alpha'\sigma_s), A_s' \rangle \in f_S$.

Since $A_s = A_e$, $\theta_s = \theta_e$, and $\sigma_s = \sigma_e$, it follows that

- $\text{DEL}(T, A_e, \alpha\sigma_e) = \text{DEL}(T_s, A_s, \alpha'\sigma_s)$.
- $N(c) \in \text{ADD}(T, A_e, \alpha\sigma_e)\theta_e$ if and only if $N(c), N^n(c) \in \text{ADD}(T_s, A_s, \alpha'\sigma_s)\theta_s$.
- $P(c_1, c_2) \in \text{ADD}(T, A_e, \alpha\sigma_e)\theta_e$ if and only if $P(c_1, c_2), P^n(c_1, c_2) \in \text{ADD}(T_s, A_s, \alpha'\sigma_s)\theta_s$.

As a consequence, since $\text{ADD}(T, A_e, \alpha\sigma_e)\theta_e$ is T -consistent, then we have $\text{ADD}(T_s, A_s, \alpha'\sigma_s)\theta_s$ is T_s -consistent. Moreover, because A_s is T_s -consistent, $\text{ADD}(T_s, A_s, \alpha'\sigma_s)\theta_s$ is T_s -consistent, and also considering how A_s' is constructed, we then have A_s' is T_s -consistent. Thus we have $\langle \langle A_s, m_s \rangle, \alpha'\sigma_s, \langle A_s', m_s' \rangle \rangle \in \text{TELL}_{f_S}$, and we also have

$$\langle A_s, m_s, \mathbf{pick} Q(\vec{p}).\alpha'(\vec{p}); \mathbf{pick} \text{true}.\alpha_e^T() \rangle \xrightarrow{\alpha'\sigma_s, f_S} \langle A_s', m_s', \mathbf{pick} \text{true}.\alpha_e^T() \rangle.$$

It is easy to see that we have

$$\langle A_s', m_s', \mathbf{pick} \text{true}.\alpha_e^T() \rangle \xrightarrow{\alpha_e^T \sigma_s', f_S} \langle A_s'', m_s'', \varepsilon \rangle$$

where

- $\langle A_s'', m_s'', \varepsilon \rangle$ is a final state
- σ_s' is empty legal parameter assignment (because α_e^T is 0-ary action).

- $m_s'' = m_e''$, (due to the fact that α_e^T does not involve any service call (i.e., $m_s'' = m_e''$) and $m_s' = m_e'$).

Additionally, by the definition of κ_E , we have $\delta_s'' = \kappa_E(\delta_e'')$ as the rest of the program to be executed (because $\langle A_s'', m_s'', \varepsilon \rangle$ is a final state). Hence, we have

$$\langle A_s, m_s, \delta_s \rangle \Rightarrow_s \langle A_s', m_s', \delta_s' \rangle \Rightarrow_s \langle A_s'', m_s'', \delta_s'' \rangle$$

To complete the proof, we obtain $A_s'' = A_e''$ simply as a consequence of the following facts:

1. $A_s = A_e$;
2. $\text{DEL}(T, A_e, \alpha\sigma_e) = \text{DEL}(T_s, A_s, \alpha'\sigma_s)$.
3. $N(c) \in \text{ADD}(T, A_e, \alpha\sigma_e)\theta_e$ if and only if $N(c), N^n(c) \in \text{ADD}(T_s, A_s, \alpha'\sigma_s)\theta_s$.
4. $P(c_1, c_2) \in \text{ADD}(T, A_e, \alpha\sigma_e)\theta_e$ if and only if $P(c_1, c_2), P^n(c_1, c_2) \in \text{ADD}(T_s, A_s, \alpha'\sigma_s)\theta_s$.
5. By Lemma 5.58, we have $N(c) \in A_e''$ if and only if either
 - $N(c) \in \text{ADD}(T, A_e, \alpha\sigma_e)\theta_e$, or
 - $N(c) \in (A_e \setminus \text{DEL}(T, A_e, \alpha\sigma_e))$ and there does not exists $B(c) \in \text{ADD}(T, A_e, \alpha\sigma_e)\theta_e$ such that $T \models N \sqsubseteq \neg B$;
6. By Lemma 5.59, we have $N(c') \in A_s''$ if and only if N is not in the vocabulary of TBox T^n and either
 - $N(c') \in \text{ADD}(T_s, A_s, \alpha'\sigma_s)\theta_s$, or
 - $N(c') \in (A_s \setminus \text{DEL}(T_s, A_s, \alpha'\sigma_s))$ and there does not exists $B(c') \in \text{ADD}(T_s, A_s, \alpha'\sigma_s)\theta_s$ such that $T \models N \sqsubseteq \neg B$.
7. α_e^T flushes all ABox assertions made by using $\text{VOC}(T^n)$.

The other direction of bisimulation relation can be proven in a similar way. \square

Having Lemma 5.60 in hand, we can easily show that given an E-GKAB, its transition system is S-bisimilar to the transition of its corresponding S-GKAB that is obtained via the translation τ_E .

Lemma 5.61. *Given an E-GKAB \mathcal{G} , we have $\Upsilon_{\mathcal{G}}^{fE} \sim_{\text{so}} \Upsilon_{\tau_E(\mathcal{G})}^{fS}$*

Proof. Let

1. $\mathcal{G} = \langle T, A_0, \Gamma, \delta_e \rangle$ and $\Upsilon_{\mathcal{G}}^{fE} = \langle \Delta, T, \Sigma_e, s_{0e}, \text{abox}_e, \Rightarrow_e \rangle$,
2. $\tau_E(\mathcal{G}) = \langle T_s, A_0, \Gamma_s, \delta_s \rangle$ and $\Upsilon_{\tau_E(\mathcal{G})}^{fS} = \langle \Delta, T_s, \Sigma_s, s_{0s}, \text{abox}_s, \Rightarrow_s \rangle$

We have that $s_{0e} = \langle A_0, m_e, \delta_e \rangle$ and $s_{0s} = \langle A_0, m_s, \delta_s \rangle$ where $m_e = m_s = \emptyset$. By the definition of κ_E and τ_E , we also have $\delta_s = \kappa_E(\delta_e)$. Hence, by Lemma 5.60, we have $s_{0e} \sim_{\text{so}} s_{0s}$. Therefore, by the definition of S-Bisimulation, we have $\Upsilon_{\mathcal{G}}^{fE} \sim_{\text{so}} \Upsilon_{\tau_E(\mathcal{G})}^{fS}$. \square

Having all of the ingredients in hand, we are now ready to show that the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ properties over E-GKABs can be recast as verification over S-GKAB as follows.

Theorem 5.62. *Given an E-GKAB \mathcal{G} and a closed $\mu\mathcal{L}_A^{\text{EQL}}$ property Φ ,*

$$\Upsilon_{\mathcal{G}}^{fE} \models \Phi \text{ iff } \Upsilon_{\tau_E(\mathcal{G})}^{fS} \models t_{\text{dup}}(\Phi)$$

Proof. By Lemma 5.61, we have that $\Upsilon_{\mathcal{G}}^{fE} \sim_{\text{so}} \Upsilon_{\tau_E(\mathcal{G})}^{fS}$. Hence, by Lemma 5.42, we have that the claim is proven. \square

5.3.4 Putting It All Together: Verification of I-GKABs

To sum up, we state the result of I-GKABs verification as follows:

Theorem 5.63. *Verification of $\mu\mathcal{L}_A^{\text{EQL}}$ properties over I-GKABs can be recast as verification over S-GKABs.*

Proof. As a consequence of Theorems 5.35, 5.54 and 5.62, we essentially show that the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ properties over I-GKABs can be recast as verification over S-GKABs since we can recast the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ properties over B-GKABs, C-GKABs, and E-GKABs as verification over S-GKABs. \square

From Theorems 4.54 and 5.63, we get our next major result that verification of all inconsistency-aware variants of GKABs introduced in Section 5.2 can be compiled into verification of KABs by concatenating the two translations from I-GKABs to S-GKABs, and then to KABs.

Theorem 5.64. *Verification of $\mu\mathcal{L}_A^{\text{EQL}}$ properties over I-GKABs can be recast as verification over KABs.*

Proof. The proof is easily obtained from the Theorems 4.54 and 5.63, since by Theorem 5.63 we can recast the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ over I-GKABs as verification over S-GKABs and then by Theorem 4.54 we can recast the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ over S-GKABs as verification over KABs. Thus combining those two ingredients, we can reduce the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ over I-GKABs into the corresponding verification of $\mu\mathcal{L}_A^{\text{EQL}}$ over KABs. \square

5.3.5 Verification of Run-bounded I-GKABs

We now aim to show that the reductions from I-GKABs to S-GKABs preserve run-boundedness.

Lemma 5.65. *Let \mathcal{G} be a B-GKAB and $\tau_B(\mathcal{G})$ be its corresponding S-GKAB. We have \mathcal{G} is run-bounded if and only if $\tau_B(\mathcal{G})$ is run-bounded.*

Proof. Let

1. $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ and $\Upsilon_{\mathcal{G}}^{f_B}$ be its transition system.
2. $\Upsilon_{\tau_B(\mathcal{G})}^{f_S}$ be the transition system of $\tau_B(\mathcal{G})$.

The proof is easily obtained due to the following facts:

- the translation τ_B essentially only appends each action invocation in δ with some additional programs to manage inconsistency.
- the actions introduced to manage inconsistency never inject new constants, but only remove facts causing inconsistency,
- by Lemma 5.34, we have that $\Upsilon_{\mathcal{G}}^{f_B} \sim_{\text{I}} \Upsilon_{\tau_B(\mathcal{G})}^{f_S}$. Thus, basically they are equivalent modulo intermediate states (states containing $\text{State}(\text{temp})$).

\square

Lemma 5.66. *Let \mathcal{G} be a C-GKAB and $\tau_C(\mathcal{G})$ be its corresponding S-GKAB. We have \mathcal{G} is run-bounded if and only if $\tau_C(\mathcal{G})$ is run-bounded.*

Proof. Similar to the proof of Lemma 5.65 but using the S-Bisimulation. \square

Lemma 5.67. *Let \mathcal{G} be a E-GKAB and $\tau_E(\mathcal{G})$ be its corresponding S-GKAB. We have \mathcal{G} is run-bounded if and only if $\tau_E(\mathcal{G})$ is run-bounded.*

Proof. Similar to the proof of Lemma 5.65 but using the S-Bisimulation. \square

Finally, we show the result on the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ properties over run-bounded I-GKABs as follows.

Theorem 5.68. *Verification of $\mu\mathcal{L}_A^{\text{EQL}}$ properties over run-bounded I-GKABs is decidable, and reducible to standard μ -calculus finite-state model checking.*

Proof. By Lemmas 5.65 to 5.67, the translation from I-GKABs to S-GKABs preserves run-boundedness. Thus, the claim follows by combining Theorem 5.63 and Theorem 4.56. \square

5.4 Back From Standard to Inconsistency-aware GKABS

So far we have seen how we can reduce the verification of I-GKABs into S-GKABs. In this section we show the other direction of reduction, i.e., we show that we can also recast the verification of S-GKABs into I-GKABs. In particular, we show how we reduce the verification of S-GKABs into B-GKABs. The reductions from S-GKABs into C-GKABs and E-GKABs can be done similarly. The following sections are then organized as follows: first we explain how we translate S-GKABs into B-GKABs. Then, we show that the transition system of S-GKABs and B-GKABs that is obtained from our translation are S-bisimilar. As a consequence, utilizing the result in Lemma 5.42, we show that we can reduce the verification of S-GKABs into B-GKABs.

The main challenge of this reduction is how to make B-GKABs mimic the standard execution semantics such that they stop evolving (instead of doing the repair) when there is an inconsistency. To deal with this, the core strategy of the translation from S-GKABs into B-GKABs is as follows:

1. We throw away all of the negative inclusion assertions as well as the functionality assertions from the current TBox and keep only the positive inclusion assertion. The purpose of this step is to avoid the changes by the repair mechanism when there is an inconsistency. Note that any ABox will be consistent with the TBox that has only positive inclusion assertions.
2. We delegate the inconsistency check to a certain action that checks the violation of negative inclusion assertions as well as the functionality assertions. To do this, we exploit the fact that the inconsistency can be checked through query answering.

Before defining the translation from S-GKABs into B-GKABs, we first introduce the translation for program in S-GKABs. Particularly, we define a translation function κ_{sb} that essentially concatenates each action invocation with an action invocation that checks the inconsistency. Additionally, the translation function κ_{sb} also serves as a one-to-one correspondence (bijection) between the original and the translated program (as well as between the sub-program).

Program Translation
 κ_{sb}

Definition 5.69 (Program Translation κ_{sb}). Given a set of actions Γ , a program δ over Γ , and a TBox T , we define a *translation* κ_{sb} which translates a program into a program inductively as follows:

$$\begin{aligned}
\kappa_{sb}(\mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p})) &= \mathbf{pick} \ Q(\vec{p}).\alpha'(\vec{p}); \mathbf{pick} \ \neg Q_{\text{unsatECQ}}^T.\alpha_{\perp}() \\
\kappa_{sb}(\varepsilon) &= \varepsilon \\
\kappa_{sb}(\delta_1|\delta_2) &= \kappa_{sb}(\delta_1)|\kappa_{sb}(\delta_2) \\
\kappa_{sb}(\delta_1;\delta_2) &= \kappa_{sb}(\delta_1);\kappa_{sb}(\delta_2) \\
\kappa_{sb}(\mathbf{if} \ \varphi \ \mathbf{then} \ \delta_1 \ \mathbf{else} \ \delta_2) &= \mathbf{if} \ \varphi \ \mathbf{then} \ \kappa_{sb}(\delta_1) \ \mathbf{else} \ \kappa_{sb}(\delta_2) \\
\kappa_{sb}(\mathbf{while} \ \varphi \ \mathbf{do} \ \delta) &= \mathbf{while} \ \varphi \ \mathbf{do} \ \kappa_{sb}(\delta)
\end{aligned}$$

where

- action $\alpha'(\vec{p})$ is obtained from $\alpha(\vec{p}) \in \Gamma$, such that

$$\text{EFF}(\alpha') = \text{EFF}(\alpha) \cup \{\mathbf{true} \rightsquigarrow \mathbf{add} \ \{\text{State}(\text{temp})\}\}$$

- Q_{unsatECQ}^T is a boolean Q-UNSAT-ECQ over T (similar to Q-UNSAT-FOL in Definition 2.43) that is used to check the inconsistency. It will be evaluated to true if the ABox is T -inconsistent.
- α_{\perp} is a 0-ary action of the form

$$\alpha_{\perp}() : \{\mathbf{true} \rightsquigarrow \mathbf{del} \ \{\text{State}(\text{temp})\}\}$$

■

Finally, to compile an S-GKAB into the corresponding B-GKAB, we define a translation τ_{sb} that, given an S-GKAB, generates a B-GKAB as follows.

Translation from
S-GKAB to B-GKAB

Definition 5.70 (Translation from S-GKAB to B-GKAB). We define a translation τ_{sb} that, given an S-GKAB $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$, generates a B-GKAB $\tau_{sb}(\mathcal{G}) = \langle T_p, A_0, \Gamma' \cup \{\alpha_{\perp}\}, \delta' \rangle$, where

- T_p is the positive inclusion assertions of T (see Definition 2.12),
- Γ' is obtained from Γ as follows: for each $\alpha \in \Gamma$, we have $\alpha' \in \Gamma$, where $\text{EFF}(\alpha') = \text{EFF}(\alpha) \cup \{\mathbf{true} \rightsquigarrow \mathbf{add} \ \{\text{State}(\text{temp})\}\}$
- α_{\perp} is an action of the form $\alpha_{\perp}() : \{\mathbf{true} \rightsquigarrow \mathbf{del} \ \{\text{State}(\text{temp})\}\}$,
- $\delta' = \kappa_{sb}(\delta)$.

■

Essentially, the translation above only keeps the positive inclusion assertions in order to prevent the repair. Moreover, it delegates the inconsistency checks into query answering.

To the aim of reducing the verification of S-GKABs into B-GKABs, in the following we show that the transition system of an S-GKAB is S-bisimilar to the transition system of its corresponding B-GKAB that is obtained via the translation τ_{sb} .

Lemma 5.71. *Let \mathcal{G} be an S-GKAB with transition system $\Upsilon_{\mathcal{G}}^{fs}$, and let $\tau_{sb}(\mathcal{G})$ be a B-GKAB with transition system $\Upsilon_{\tau_{sb}(\mathcal{G})}^{fB}$ obtained through τ_{sb} . Consider a state $\langle A_s, m_s, \delta_s \rangle$ of $\Upsilon_{\mathcal{G}}^{fs}$ and a state $\langle A_b, m_b, \delta_b \rangle$ of $\Upsilon_{\tau_{sb}(\mathcal{G})}^{fB}$. If $A_s = A_b$, $m_s = m_b$ and $\delta_s = \kappa_{sb}(\delta_b)$, then $\langle A_s, m_s, \delta_s \rangle \sim_{\text{SO}} \langle A_b, m_b, \delta_b \rangle$.*

Proof. Similar line of reasoning as in the proof of Lemmas 5.33, 5.52 and 5.60 can be applied here. The important different are as follows:

- Each state in the transition system of B-GKAB is always consistent, because we throw away all negative inclusion assertions as well as functionality assertions from the TBox and only keep the positive inclusion assertions. As a consequence, the repair mechanism in B-GKAB will not change anything.
- Each action invocation in the program in the given S-GKAB is translated in such a way that it will always be followed by the action invocation **pick** $\neg Q_{\text{unsatECQ}}^T \cdot \alpha_{\perp}()$. Note that Q_{unsatECQ}^T will be evaluated to true when the corresponding ABox is T -inconsistent. Therefore, the inconsistency check is basically delegated to the evaluation of the query that acts as the guard of the action α_{\perp} and it is triggered after each action execution. Moreover, the action α_{\perp} will not be executed if the previous action execution leads into an inconsistent state w.r.t. the TBox T . Thus, it is easy to see that when an action execution in S-GKAB is blocked because it leads into a T -inconsistent state, then the corresponding action execution in B-GKAB will not lead into a new state without $\text{State}(\text{temp})$ as well. However, when an execution in S-GKAB leads into a new T -consistent state, the corresponding action execution in B-GKAB will be followed by the execution of α_{\perp} and it leads into a new state without $\text{State}(\text{temp})$. □

Having Lemma 5.71 in hand, we can easily show that given an S-GKAB, its transition system is S-bisimilar to the transition of its corresponding B-GKAB that is obtained via the translation τ_{sb} as follows.

Lemma 5.72. *Given an S-GKAB \mathcal{G} , we have $\Upsilon_{\mathcal{G}}^{fs} \sim_{\text{so}} \Upsilon_{\tau_{sb}(\mathcal{G})}^{fB}$*

Proof. Let

1. $\mathcal{G} = \langle T, A_0, \Gamma, \delta_s \rangle$, and $\Upsilon_{\mathcal{G}}^{fs} = \langle \Delta, T, \Sigma_s, s_{0s}, \text{abox}_s, \Rightarrow_s \rangle$,
2. $\tau_{sb}(\mathcal{G}) = \langle T_b, A_0, \Gamma_b, \delta_b \rangle$, and $\Upsilon_{\tau_{sb}(\mathcal{G})}^{fB} = \langle \Delta, T_b, \Sigma_b, s_{0b}, \text{abox}_b, \Rightarrow_b \rangle$.

We have that $s_{0s} = \langle A_0, m_s, \delta_s \rangle$ and $s_{0b} = \langle A_0, m_b, \delta_b \rangle$ where $m_s = m_b = \emptyset$. By the definition of κ_{sb} and τ_{sb} , we also have $\delta_b = \kappa_{sb}(\delta_s)$. Hence, by Lemma 5.71, we have $s_{0s} \sim_{\text{so}} s_{0b}$. Therefore, by the definition of S-bisimulation, we have $\Upsilon_{\mathcal{G}}^{fs} \sim_{\text{so}} \Upsilon_{\tau_{sb}(\mathcal{G})}^{fB}$. □

Finally, we now able to show that the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ formulas over S-GKABs can be recast as verification of $\mu\mathcal{L}_A^{\text{EQL}}$ formulas over B-GKABs as follows.

Theorem 5.73. *Given an S-GKAB \mathcal{G} and a closed $\mu\mathcal{L}_A^{\text{EQL}}$ property Φ ,*

$$\Upsilon_{\mathcal{G}}^{fs} \models \Phi \text{ iff } \Upsilon_{\tau_{sb}(\mathcal{G})}^{fB} \models t_{sb}(\Phi)$$

Proof. By Lemma 5.72, we have that $\Upsilon_{\mathcal{G}}^{fs} \sim_{\text{so}} \Upsilon_{\tau_{sb}(\mathcal{G})}^{fB}$. Hence, by Lemma 5.42, it is easy to see that the claim is proven. □

5.5 Discussion: Extended Inconsistency-Aware Golog-KABs

In the following we elaborate some possible extensions of I-GKABs. First, we discuss an I-GKABs extension that enables us to keep track some information regarding inconsistencies and hence it gives us finer-grained insights concerning inconsistencies. Second, we elaborate a possibility to allow dynamic selection of repair mechanisms and hence enable us incorporate several repair mechanisms within one system.

5.5.1 Tracking Inconsistencies

As we have seen, I-GKABs (B-GKABs, C-GKABs, E-GKABs) enhance GKABs with inconsistency-handling mechanisms by adopting repair-based semantics. However, despite dealing with possible repairs when some action step produces a T -inconsistent ABox, they do not explicitly track whether a repair has been actually enforced, nor do they provide finer-grained insights about which TBox assertions were involved in the inconsistency. As a brief discussion, here we elaborate a possible extension of I-GKABs so as to equip the transition system of I-GKABs with these additional information. We will also see that such information enable us to have a more fine-grained analysis over the system evolution. Furthermore, we also give an intuition that the verification of GKABs with such extended inconsistency-aware semantics can be reduced to the corresponding verification of S-GKABs.

The idea of extending the inconsistency-aware semantics for GKABs is elaborated step by step as follows:

1. We assume w.l.o.g. that Δ_0 contains one distinguished constant per TBox assertion in T ,
2. We introduce a function LABEL, that, given a TBox assertion, returns the corresponding constant.
3. We then define the set $\text{VIOL}(A, T)$ of constants labeling TBox assertions in T violated by A , as follows:

$$\begin{aligned} \text{VIOL}(A, T) = & \{d \in \Delta \mid (\text{funct } Z) \in T_f, d = \text{LABEL}((\text{funct } Z)) \text{ and} \\ & A \models \exists xyz. q_{\text{unsat}}^f((\text{funct } Z), x, y, z)\} \cup \\ & \{d \in \Delta \mid B_1 \sqsubseteq \neg B_2 \in T_n, d = \text{LABEL}(B_1 \sqsubseteq \neg B_2) \text{ and} \\ & A \models \text{rew}(\exists x. q_{\text{unsat}}^n(B_1 \sqsubseteq \neg B_2, x), T)\} \cup \\ & \{d \in \Delta \mid R_1 \sqsubseteq \neg R_2 \in T_n, d = \text{LABEL}(R_1 \sqsubseteq \neg R_2) \text{ and} \\ & A \models \text{rew}(\exists x. q_{\text{unsat}}^n(R_1 \sqsubseteq \neg R_2, x), T)\} \end{aligned}$$

4. We then employ the information provided by the set $\text{VIOL}(A, T)$ to decorate the states that is produced in each transition. This is done by utilizing a fresh concept Viol that keeps track of the labels of violated TBox assertions. Formally, we adjust the definitions of b-repair filter (Definition 5.6), c-repair filter (Definition 5.9), and e-repair filter (Definition 5.12) as follows:
 - A *B-repair Filter* f_B is a relation that consists of tuples of the form $\langle A, F^+, F^-, A' \rangle$ such that $A' \in \text{B-REP}(T, (A \setminus F^-) \cup F^+)$ and $A'' = A' \cup \{\text{Viol}(d) \mid d \in \text{VIOL}((A \setminus F^-) \cup F^+, T)\}$.

- A *C-repair Filter* f_C is a relation that consists of tuples of the form $\langle A, F^+, F^-, A'' \rangle$ such that $A' = \text{C-REP}(T, (A \setminus F^-) \cup F^+)$ and $A'' = A' \cup \{\text{Viol}(d) \mid d \in \text{VIOL}((A \setminus F^-) \cup F^+, T)\}$.
- A *B-evol Filter* f_E is a relation that consists of tuples of the form $\langle A, F^+, F^-, A'' \rangle$ such that $A' = \text{EVOL}(T, A, F^+, F^-)$, F^+ is T -consistent, and $A'' = A' \cup \{\text{Viol}(d) \mid d \in \text{VIOL}((A \setminus F^-) \cup F^+, T)\}$.

Additionally, notice that we also need to flush away each existing ABox assertion made by the concept *Viol* whenever we want to generate a new state. This is necessary in order to make sure that the information about violated TBox assertions from the previous states are not augmented to the new states. It is easy to see that we can achieve such purpose by modifying the definition of the filters above a little bit.

We have seen how we can decorate each state in the transition systems of I-GKABs such that they contain information about violated TBox assertion. Now, with this machinery in hand, observe that we can do finer-grained analysis over the system evolution by exploiting the concept *Viol*. For instance we can now verify the following $\mu\mathcal{L}_A^{\text{EQL}}$ properties:

- $\nu Z.(\neg \exists l. \text{Viol}(l)) \wedge [\neg]Z$. It says that no state of the system is violating the TBox constraints;
- $\nu Z.(\forall l. \text{Viol}(l) \rightarrow (\mu Y.(\nu W. \neg \text{Viol}(l) \wedge [\neg]W) \vee \langle \neg \rangle Y)) \wedge [\neg]Z$. It says that, in all states, whenever a certain TBox assertion t is violated, there exists a run that reaches a state starting from which t will never be violated anymore.

We now proceed to give the intuition that the verification of I-GKABs (B-GKABs, C-GKABs, and E-GKABs) with such kind of extension can be reduced to the corresponding verification of S-GKABs as follows:

1. Since we can detect the violated TBox assertions through query answering, we can simply construct an action in which each of its effects detects the violation of a particular TBox assertion, and then when a certain TBox assertion is violated, this action adds the corresponding assertion made by the concept *Viol* and the corresponding label of the violated TBox assertion.
2. We concat each action invocation with the action that marks the violated TBox assertions and then we concat them with the repair program. With this approach, we can simulate the computation of extended inconsistency-aware GKABs inside S-GKABs.
3. As for translating the temporal properties, similar approach can be followed. For the case of extended C-GKABs and E-GKABs we might need to triplicate the modal operator instead of just duplicating it.

Last, note that it is easy to see that our extended inconsistency-aware semantics can easily capture S-GKABs. Similar approach as in Section 5.4 can be followed.

5.5.2 Dynamic Selection of Repair Mechanisms

In the I-GKABs framework that has been presented so far, each of them only employ one kind of repair mechanism when there is an inconsistency (e.g., B-GKABs always apply b-repair when there is an inconsistency). However, it might be desirable to

employ different kind of repair mechanisms in a different situation within a system. For example, when specifying the program, the designer of the system might know that in case there is an inconsistency after the execution of a particular action, it is better to apply a certain repair mechanism instead of another repair mechanisms. In general, it might also desirable to use different way of updating the ABox for different action.

To deal with this, we can extend our Golog program (cf. Definition 4.1) such that we can have *annotated atomic action invocation* that is atomic action invocation annotated with the desired way of updating the ABox as well as the preferred repair mechanism. Technically, it is the usual atomic action invocation annotated with a filter relation written as follows:

$$\mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p}) : \mathbf{f}$$

where \mathbf{f} is the desired filter relation. Furthermore, we can refine the notion of program execution relation (cf. Definition 4.15) by adding the following:

- $\langle A, m, \mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p}) : \mathbf{f} \rangle \xrightarrow{\alpha\sigma, f} \langle A', m', \varepsilon \rangle$
if $\langle A, m, \mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p}) \rangle \xrightarrow{\alpha\sigma, \mathbf{f}} \langle A', m', \varepsilon \rangle$

Note that the additional condition for the program execution relation above updates A into A' by employing filter \mathbf{f} instead of the given filter f . Essentially we define the relation $\xrightarrow{\alpha\sigma, f}$ by also utilizing the relation $\xrightarrow{\alpha\sigma, \mathbf{f}}$. As an example, one might specify the following atomic action invocation:

$$\mathbf{pick} \ \text{true}.\alpha() : f_C$$

which requires that the update by the action α should be done using the c-repair filter f_C .

Regarding verification, it can be shown that the verification of I-GKABs with such kind of extension can be reduced to verification of S-GKABs as follows. The intuition is that we just need to translate each annotated action invocation into an action invocation that is concatenated with the corresponding program that is suitable with the preferred way of updating the ABox. For instance, to simulate $\mathbf{pick} \ \text{true}.\alpha() : f_C$ inside S-GKAB, we can concat $\mathbf{pick} \ \text{true}.\alpha()$ with c-repair program.

Now, instead of just annotating some particular atomic action invocations, one might want to annotate a program with a preferred way of updating an ABox. Thus, we can extend our Golog program further with the following construct:

$$\delta : \mathbf{f}$$

where \mathbf{f} is a filter relation and δ is a golog program that does not contain any annotated atomic action invocation. We can then extend the program execution relation as follows:

$$\langle A, m, \delta : \mathbf{f} \rangle \xrightarrow{\alpha\sigma, f} \langle A', m', \delta' : \mathbf{f} \rangle \text{ if } \langle A, m, \delta \rangle \xrightarrow{\alpha\sigma, \mathbf{f}} \langle A', m', \delta' \rangle$$

However, notice that the construct $\delta : \mathbf{f}$ essentially can be translated into the previous setting (i.e., the setting with annotated atomic action invocation) by translating each atomic action invocation $\mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p})$ within $\delta : \mathbf{f}$ into an annotated atomic action invocation $\mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p}) : \mathbf{f}$. Therefore, in the end the verification in this setting can also be reduced to the verification of S-GKABs.

EMBRACING CONTEXTS INTO GOLOG-KAB_s

In GKABs (as well as in KABs), the intensional knowledge about the domain, expressed in terms of a DL TBox, is assumed to be fixed along the evolution of the system, i.e., independent of the actual state. However, this assumption is in general too restrictive, since specific knowledge might hold or be applicable only in specific, *context-dependent* circumstances. Ideally, one should be able to form statements that are known to be true in certain cases, but not necessarily in all. For instance, in our simple order processing scenario (cf. Section 2.1 and example 4.3), in the normal situation an employee should only act as either a designer or an assembler. However, during the peak season, a designer might also work as an assembler. In addition, the needs of having flexible business processes that are able to adapt themselves according to the situation (context) also has been identified in the area of business process modeling [162, 153]. For example, in our simple order processing scenario, under the peak season, the company might prefer to outsource the quality control operation instead of performing such action by themselves.

In this chapter, we enrich GKABs with contextual information, making use of different context dimensions. On the one hand, context is determined by the environment using context-changing actions that make use of the current state of the KB and the current context. On the other hand, it affects the set of TBox assertions that are relevant at each time point, and that have to be considered when processing queries posed over GKABs.

We follow here the approach of [19, 80], and introduce *contextualized TBoxes*, in which each inclusion assertion is adorned with context information that determines under which circumstances the inclusion assertion is considered to hold. The relation among contexts is described by means of a lattice in [19] and by means of a directed acyclic graph in [80]. In our case, we represent context using a finite set of context dimensions, each characterized by a finite set of domain values that are organized in a tree structure. If for a context dimension d , a value v_2 is placed below v_1 in the tree (i.e., v_2 is a descendant of v_1), then the context associated to v_1 is considered to be more general than the one for v_2 , and hence whenever context dimension d is in value v_2 , it is also in value v_1 .

Starting from this representation of contexts, we enrich GKABs towards *Context-Sensitive GKABs* (CSGKABs), by representing the intensional information about the domain using a contextualized TBox, in place of an ordinary one. Moreover, the action component of GKABs, which specifies how the states of the system evolve, is extended in CSGKABs with *context changing actions*. Such actions determine values for context dimensions in the new state, based on the data and the context in the current state. In addition, also regular state-changing actions can query, besides the state, also the context, and hence be enabled or disabled according to the context. Notably, we show

that verification of a very rich temporal logic, which can be used to query the system evolution, contexts, and data, is decidable for run-bounded CSGKABs.

For the setting, as in KABs and GKABs, in the following we use $DL-Lite_{\mathcal{A}}$ for expressing KBs and we also do not distinguish between objects and values (thus we drop attributes). Moreover we make use of a countably infinite set Δ of constants, which intuitively denotes all possible values in the system. Additionally, we consider a finite set of distinguished constants $\Delta_0 \subset \Delta$, and a finite set \mathcal{F} of *function symbols* that represents *service calls*, which abstractly account for the injection of fresh values (constants) from Δ into the system.

The core results in this chapter are published in [68, 67]

6.1 Context Formalization

Following [140], we formalize context as a mathematical object. Basically, we follow the approach in [166] of contextualizing knowledge bases by adopting the metaphor of considering context as a box [42, 112]. Specifically, this means that the knowledge represented by the TBox (together with the ABox) in a certain context is affected by the values of parameters used to characterize the context itself.

*Tree-shaped Value
Domain*

Definition 6.1 (Tree-shaped Value Domain). Given a variable d , a *tree-shaped finite value domain* of d is a pair $\langle Dom(d), \prec_d \rangle$ where:

- $Dom(d)$ is a finite set of domain values, and
- \prec_d is a binary relation between values in $Dom(d)$

and it holds that

1. There exists exactly one value $v \in Dom(d)$ such that there does not exist $v' \in Dom(d)$ and $v \prec_d v'$ (in this case we say that v is a *root*),
2. For each $v \in Dom(d)$, if v is not a root, then there exists exactly one value $v' \in Dom(d)$ such that $v \prec_d v'$, and
3. There is no cycle (i.e., there does not exist v_1, v_2, \dots, v_n such that $v_1 \prec_d v_2, v_2 \prec_d v_3, \dots, v_n \prec_d v_1$).

■

In the definition above, intuitively the condition 1 to 3 impose that the binary relation \prec_d relates the values in $Dom(d)$ such that they form a tree structure. As notation, we denote the domain value in the root of the tree with \top_d . Intuitively, \top_d is the most general value in the tree-shaped value hierarchy of $Dom(d)$.

We now proceed to define the notion of context dimension and context dimension assignment which are the crucial ingredients for formalizing the notion of context.

Context Dimension

Definition 6.2 (Context Dimension). A *context dimension* is a variable d that has its own *tree-shaped finite value domain* $\langle Dom(d), \prec_d \rangle$. ■

*Context Dimension
Assignment*

Definition 6.3 (Context Dimension Assignment). Let d be a context dimension with a tree-shaped finite value domain $\langle Dom(d), \prec_d \rangle$. A *context dimension assignment*, denoted by $[d \rightsquigarrow v]$, is the assignment of value $v \in Dom(d)$ into the context dimension d . ■

Intuitively, a *context dimension assignment* $[d \rightsquigarrow v]$ denotes the fact that a context dimension d is in value v .

From this moment, except for Section 6.6, for the technical development of this chapter we fix a set

$$\mathbb{D} = \{d_1, \dots, d_n\}$$

of context dimensions. Each context dimension $d_i \in \mathbb{D}$ comes with its own tree-shaped finite value domain $\langle \text{Dom}(d_i), \prec_{d_i} \rangle$, where $\text{Dom}(d_i)$ represents the finite set of domain values, and \prec_{d_i} represents the predecessor relation forming the tree.

The notion of context is then formally defined as follows:

Definition 6.4 (Context). A *context* C over a set \mathbb{D} of context dimensions is defined as a set $\{[d_1 \rightsquigarrow v_1], \dots, [d_n \rightsquigarrow v_m]\}$ of context dimension assignments such that for each context dimension $d \in \mathbb{D}$, there exists exactly one assignment $[d \rightsquigarrow v] \in C$. ■ Context

Informally, a context represents a particular situation and is characterized by the assignment of a particular value into each context dimension.

Example 6.5. Recall our simple order processing scenario in Example 4.3. Here we extend this running example into the case where some contextual information come into play. Consider the following situations:

- Under a certain processing plan or during a particular season, the company might prefer to perform a certain operation compared to the other operation (e.g., During peak season, instead of doing the quality check by themselves, the company might outsource the operation).
- Within a specific season or when a particular processing plan is applied, our domain knowledge might changes.

To model this situation, in this scenario we consider the set of context dimensions $\mathbb{D} = \{\text{PP}, \text{S}\}$, where PP stands for *processing plan*, and S stands for *season*. Both context dimensions PP and S are used later to characterized some contexts (some particular situations). We then define $\text{Dom}(\text{PP})$ as well as $\text{Dom}(\text{S})$ as follows:

- $\text{Dom}(\text{PP}) = \{\text{WE}, \text{ME}, \text{RE}, \text{N}, \text{AP}\}$, where
 1. WE stands for *worker efficiency*,
 2. ME stands for *material efficiency*,
 3. RE stands for *resource efficiency*,
 4. N stands for *normal processing plan*, and
 5. AP stands for *any processing plan*.

and

1. $\text{WE} \prec_{\text{PP}} \text{RE}$,
2. $\text{ME} \prec_{\text{PP}} \text{RE}$,
3. $\text{RE} \prec_{\text{PP}} \text{AP}$,
4. $\text{N} \prec_{\text{PP}} \text{AP}$.

To give more intuition, the value domain of PP is visually described in Figure 8. As an example, $\text{WE} \prec_{\text{PP}} \text{RE}$ means that *worker efficiency* is a form of *resource efficiency*.

- $Dom(S) = \{WH, PS, LS, NS, AS\}$, where

1. WH stands for *winter holiday*,
2. PS stands for *peak season*,
3. LS stands for *low season*,
4. NS stands for *normal season*,
5. AS stands for *any season*.

and

1. $WH \prec_S PS$,
2. $PS \prec_S AS$,
3. $NS \prec_S AS$,
4. $LS \prec_S AS$.

To give more intuition, the value domain of S is visually described in Figure 9.

An example of a context over the set $\mathbb{ID} = \{PP, S\}$ of context dimensions is the context

$$C = \{[PP \rightsquigarrow N], [S \rightsquigarrow NS]\},$$

which essentially encode the context (situation) where the normal processing plan is applied and the season is normal. Technically, the context C is characterized by the assignment of the value N into PP and the value N into S .

To predicate over contexts, we introduce a *context expression language* \mathcal{L}_{cx} over \mathbb{ID} , which corresponds to propositional logic where the propositional letters are context dimension assignments over \mathbb{ID} . Formally, it is defined as follows:

Context Expression
Language

Definition 6.6 (Context Expression Language). The syntax of *context expression language* \mathcal{L}_{cx} is as follows:

$$\varphi_C ::= [d \rightsquigarrow v] \mid \varphi_C \wedge \varphi'_C \mid \neg \varphi_C$$

where $d \in \mathbb{ID}$, and $v \in Dom(d)$. ■

We call a formula expressed in \mathcal{L}_{cx} a *context expression*. For the semantics of \mathcal{L}_{cx} , we adopt the standard propositional logic semantics and the usual abbreviations. The notion of *satisfiability* and *model* are as usual.

Observe that a context $C = \{[d_1 \rightsquigarrow v_1], \dots, [d_n \rightsquigarrow v_m]\}$, being a set of (atomic) formulas in \mathcal{L}_{cx} , can be considered as a propositional theory. The semantics of value domain of each context dimension in \mathbb{ID} can also be characterized by an \mathcal{L}_{cx} theory. Specifically, we define the *value domain theory* $\Phi_{\mathbb{ID}}$ of \mathbb{ID} as an \mathcal{L}_{cx} theory below:

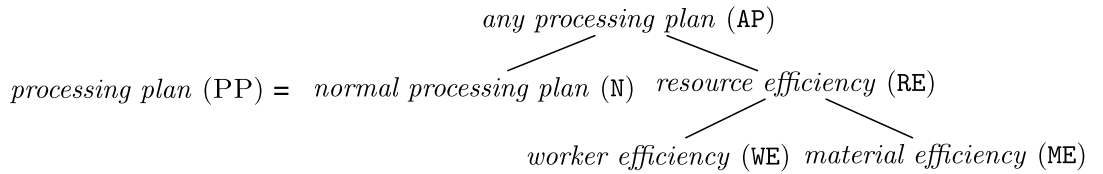


Figure 8: Value domain of the context dimension *processing plan* (PP)

Definition 6.7 (Context Dimension Value Domain Theory). We define a *value domain theory* $\Phi_{\mathbb{D}}$ of \mathbb{D} as the smallest set of context expressions satisfying the following conditions: For every context dimension $d \in \mathbb{D}$, we have:

Context Dimension
Value Domain Theory

- For all values $v_1, v_2 \in \text{Dom}(d)$ such that $v_1 \prec_d v_2$, we have that $\Phi_{\mathbb{D}}$ contains the expression $[d \rightsquigarrow v_1] \rightarrow [d \rightsquigarrow v_2]$. Intuitively, this states that the value v_2 is more general than v_1 , and hence, whenever we have $[d \rightsquigarrow v_1]$ we can infer that $[d \rightsquigarrow v_2]$.
- For all values $v_1, v_2, v \in \text{Dom}(d)$ s.t. $v_1 \prec_d v$ and $v_2 \prec_d v$, we have that $\Phi_{\mathbb{D}}$ contains the expression $[d \rightsquigarrow v_1] \rightarrow \neg[d \rightsquigarrow v_2]$. Intuitively, this expresses that sibling values v_1 and v_2 are disjoint.

■

In the following, we write $\Phi_{\mathbb{D}}$ to denote the value domain theory of \mathbb{D} .

6.2 Contextualizing Knowledge Bases

Essentially, we define a *context-sensitive knowledge base* (CKB) over the set \mathbb{D} of context dimensions as a standard DL knowledge base in which the TBox assertions are contextualized.

Definition 6.8 (Contextualized TBox). A *contextualized TBox* T_{cx} over \mathbb{D} is a finite set of assertions of the form $\langle t : \varphi \rangle$, where t is a usual TBox assertion and φ is a context expression over \mathbb{D} .

Contextualized TBox

■

Intuitively, $\langle t : \varphi \rangle$ expresses that the TBox assertion t holds in all those contexts satisfying φ , taking into account the theory $\Phi_{\mathbb{D}}$. Similar to the usual TBox, given a contextualized TBox T_{cx} , we write $\text{voc}(T_{cx})$ to denote the vocabulary of TBox T_{cx} , independently from the context. As a remark, the idea of our contextualized TBox is inspired by [80] (see the notion of V-TBox in [80]).

Definition 6.9 (Contextualized KB). A *contextualized KB* is a tuple $\langle T_{cx}, A \rangle$ where T_{cx} is a contextualized TBox and A is the usual ABox over $\text{voc}(T_{cx})$.

Contextualized KB

■

We now define the notion of a KB in context C as follows:

Definition 6.10 (KB Under the Context C). Given a CKB $\langle T_{cx}, A \rangle$ and a context C , both over \mathbb{D} , we define the *KB under the context C* as the KB $\langle T_{cx}^C, A \rangle$, where $T_{cx}^C = \{t \mid \langle t : \varphi \rangle \in T_{cx} \text{ and } C \cup \Phi_{\mathbb{D}} \models \varphi\}$. Additionally, in this case we say that T_{cx}^C is *contextualized TBox T_{cx} under the context C* .

KB Under the
Context C

■

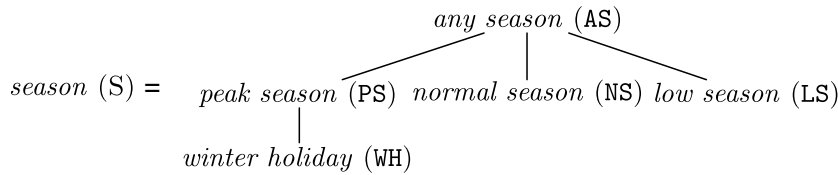


Figure 9: Value domain of the context dimension *season* (S)

Example 6.11. Continuing our example, in a normal situation, to enforce the segregation of duties, *designer* and *assembler* must be different. However, in the situation (context) where we have either *peak season* ($[S \rightsquigarrow \text{PS}]$) or the company wants to promote *worker efficiency* ($[PP \rightsquigarrow \text{WE}]$), each *designer* is also an *assembler*. In addition, the other assertions hold in any situation (context). This situation can be encoded in a contextualized TBox T_{cx} containing the following assertions:

$$\begin{aligned}
&\langle \text{ApprovedOrder} \sqsubseteq \text{Order} : [PP \rightsquigarrow \text{AP}] \wedge [S \rightsquigarrow \text{AS}] \rangle \\
&\langle \text{AssembledOrder} \sqsubseteq \text{Order} : [PP \rightsquigarrow \text{AP}] \wedge [S \rightsquigarrow \text{AS}] \rangle \\
&\langle \text{DeliveredOrder} \sqsubseteq \text{Order} : [PP \rightsquigarrow \text{AP}] \wedge [S \rightsquigarrow \text{AS}] \rangle \\
&\langle \text{ReceivedOrder} \sqsubseteq \text{Order} : [PP \rightsquigarrow \text{AP}] \wedge [S \rightsquigarrow \text{AS}] \rangle \\
&\langle \text{Designer} \sqsubseteq \text{Employee} : [PP \rightsquigarrow \text{AP}] \wedge [S \rightsquigarrow \text{AS}] \rangle \\
&\langle \text{Assembler} \sqsubseteq \text{Employee} : [PP \rightsquigarrow \text{AP}] \wedge [S \rightsquigarrow \text{AS}] \rangle \\
&\langle \text{QualityController} \sqsubseteq \text{Employee} : [PP \rightsquigarrow \text{AP}] \wedge [S \rightsquigarrow \text{AS}] \rangle \\
&\langle \text{Designer} \sqsubseteq \neg \text{Assembler} : [PP \rightsquigarrow \text{N}] \wedge ([S \rightsquigarrow \text{NS}] \vee [S \rightsquigarrow \text{LS}]) \rangle \\
&\langle \text{Designer} \sqsubseteq \text{Assembler} : [PP \rightsquigarrow \text{WE}] \vee [S \rightsquigarrow \text{PS}] \rangle \\
&\langle \text{Designer} \sqsubseteq \neg \text{QualityController} : [PP \rightsquigarrow \text{AP}] \vee [S \rightsquigarrow \text{AS}] \rangle \\
&\langle \text{Assembler} \sqsubseteq \neg \text{QualityController} : [PP \rightsquigarrow \text{AP}] \wedge [S \rightsquigarrow \text{AS}] \rangle \\
&\langle \exists \text{assembledBy}^- \sqsubseteq \text{Employee} : [PP \rightsquigarrow \text{AP}] \wedge [S \rightsquigarrow \text{AS}] \rangle \\
&\langle \exists \text{assembledBy} \sqsubseteq \text{Order} : [PP \rightsquigarrow \text{AP}] \wedge [S \rightsquigarrow \text{AS}] \rangle \\
&\langle \exists \text{designedBy}^- \sqsubseteq \text{Employee} : [PP \rightsquigarrow \text{AP}] \wedge [S \rightsquigarrow \text{AS}] \rangle \\
&\langle \exists \text{designedBy} \sqsubseteq \text{Order} : [PP \rightsquigarrow \text{AP}] \wedge [S \rightsquigarrow \text{AS}] \rangle \\
&\langle \exists \text{checkedBy}^- \sqsubseteq \text{Employee} : [PP \rightsquigarrow \text{AP}] \wedge [S \rightsquigarrow \text{AS}] \rangle \\
&\langle \exists \text{checkedBy} \sqsubseteq \text{Order} : [PP \rightsquigarrow \text{AP}] \wedge [S \rightsquigarrow \text{AS}] \rangle \\
&\langle \exists \text{hasAssemblingLoc}^- \sqsubseteq \text{Location} : [PP \rightsquigarrow \text{AP}] \wedge [S \rightsquigarrow \text{AS}] \rangle \\
&\langle \exists \text{hasAssemblingLoc} \sqsubseteq \text{Order} : [PP \rightsquigarrow \text{AP}] \wedge [S \rightsquigarrow \text{AS}] \rangle \\
&\langle \exists \text{hasDesign}^- \sqsubseteq \text{Design} : [PP \rightsquigarrow \text{AP}] \wedge [S \rightsquigarrow \text{AS}] \rangle \\
&\langle \exists \text{hasDesign} \sqsubseteq \text{Order} : [PP \rightsquigarrow \text{AP}] \wedge [S \rightsquigarrow \text{AS}] \rangle \\
&\langle (\text{funct hasAssemblingLoc}) : [PP \rightsquigarrow \text{AP}] \wedge [S \rightsquigarrow \text{AS}] \rangle \\
&\langle (\text{funct hasDesign}) : [PP \rightsquigarrow \text{AP}] \wedge [S \rightsquigarrow \text{AS}] \rangle
\end{aligned}$$

Given a CKB $\langle T_{cx}, A \rangle$, and a context $C = \{[PP \rightsquigarrow \text{N}], [S \rightsquigarrow \text{NS}]\}$, we have that $\langle T_{cx}^C, A \rangle$ is a KB under the context C where T_{cx}^C containing the same TBox assertions as in Example 2.17. As another example, consider the context $C' = \{[PP \rightsquigarrow \text{WE}], [S \rightsquigarrow \text{PS}]\}$, we have that $\langle T_{cx}^{C'}, A \rangle$ is a KB under the context C' where $T_{cx}^{C'}$ containing the same TBox assertions as in Example 2.17 except that it contains $\text{Designer} \sqsubseteq \text{Assembler}$ instead of $\text{Designer} \sqsubseteq \neg \text{Assembler}$.

6.3 Contextualizing Golog-Program

As the presence of contexts influence the condition when an action can be executed, we now lift our Golog program variant (in Definition 4.1) into Contextualized Golog program as follows:

Definition 6.12 (Contextualized Golog Program). Given a set of KAB actions Γ , a *contextualized Golog program* δ over Γ is an expression formed by the following grammar: Contextualized Golog Program

$$\delta ::= \varepsilon \mid \mathbf{pick} \langle Q(\vec{p}), \varphi_C \rangle . \alpha(\vec{p}) \mid \delta_1 | \delta_2 \mid \delta_1 ; \delta_2 \mid \\ \mathbf{if} \varphi \mathbf{then} \delta_1 \mathbf{else} \delta_2 \mid \mathbf{while} \varphi \mathbf{do} \delta$$

where:

- ε , $\delta_1 | \delta_2$, $\delta_1 ; \delta_2$, **if** φ **then** δ_1 **else** δ_2 and **while** φ **do** δ are the same as in Definition 4.1.
- **pick** $\langle Q(\vec{p}), \varphi_C \rangle . \alpha(\vec{p})$ is a *context-sensitive atomic action invocation* guarded by a DI-ECQ Q and context expression φ_C , such that $\alpha \in \Gamma$ is executable when φ_C is satisfied by the current context (taking into account the theory Φ_D), and it is executed by non-deterministically substituting its parameters \vec{p} with an answer of Q ;

■

6.4 Context-Sensitive Golog-KABs (CSGKABs)

We now enhance GKABs with context-related information, introducing in particular *Context-Sensitive GKABs* (CSGKABs), which consist of:

1. a context-sensitive knowledge base (CKB), which maintains the information of interest,
2. an action base, which characterizes the system evolution, and
3. context information that evolves over time, capturing changing circumstances.

Differently from GKABs, where the TBox is fixed a-priori and remains rigid during the evolution of the system, in CSGKABs the TBox changes depending on the current context. Alongside the evolution mechanism for data borrowed from GKABs, CSGKABs include also a progression mechanism for the context itself, giving raise to a system in which data and context evolve simultaneously.

Definition 6.13 (Context-Sensitive GKABs (CSGKABs)). A CSGKAB is a tuple $\mathcal{G}_{cx} = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$ where: Context-Sensitive GKABs (CSGKABs)

- T_{cx} is a *DL-Lite_A contextualized TBox* capturing the domain of interest.
- A_0 and Γ are as in a GKAB.
- δ is a contextualized Golog program over Γ , which characterizes the evolution of the GKAB over time, using the atomic actions in Γ .
- C_0 is the initial context over \mathbb{D} .
- Π_C is a finite set of context-evolution rules, each of which determines the configuration of the new context depending on the current context and data. Each *context-evolution rule* has the form $\langle Q, \varphi_C \rangle \mapsto C_{new}$, where:
 1. Q is a boolean ECQ over T_{cx} ,
 2. φ_C is a context expression, and

3. C_{new} is a finite set of context dimension assignments such that for each context dimension $d \in \mathbb{D}$, there exists *at most one* context dimension assignment $[d \rightsquigarrow v] \in C_{new}$. In the execution, if a context variable is not assigned by C_{new} , it maintains the assignment of the previous state. ■

Example 6.14. An example of a CSGKAB.

Continuing our running example, consider the scenario in which either during the *peak season* ($[S \rightsquigarrow PS]$) or when the company wants to promote *worker efficiency* ($[PP \rightsquigarrow WE]$), the company outsource the quality control task. To model this scenario, we specify a CSGKAB $\mathcal{G}_{cx} = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$ where T_{cx} is the same as contextualized TBox in Example 6.11, A_0 is the same as the one in Example 4.3, Γ is the same as in Example 3.4 except that we augment an additional action that essentially performs the quality control by outsourcing it. This action obtains the quality controller by calling a service GETQCSERVICE/1 and it is specified as follows:

```

outsourcingQualityCheck() : {
    [AssembledOrder(x)]  $\rightsquigarrow$ 
    add { checkedBy(x, GETQCSERVICE(x)) }
}.

```

To realize the flow of order processing in the scenario above, the initial program δ is specified as follows:

$$\delta = \textbf{while } \exists x. [\text{Order}(x)] \wedge \neg [\text{DeliveredOrder}(x)] \textbf{ do } \delta_0$$

where:

- $\delta_0 = \delta_1; \delta_2; \delta_3; \delta_4; \delta_5$
- $\delta_1 = \textbf{if } \neg [\exists x. \text{ApprovedOrder}(x)]$
 then pick $\langle \text{ReceivedOrder}(x), [PP \rightsquigarrow AP] \wedge [S \rightsquigarrow AS] \rangle. \text{approveOrder}(x)$
 else ε ,
- $\delta_2 = \textbf{pick } \langle \text{true}, [PP \rightsquigarrow AP] \wedge [S \rightsquigarrow AS] \rangle. \text{prepareOrders}()$,
- $\delta_3 = \textbf{pick } \langle \text{true}, [PP \rightsquigarrow AP] \wedge [S \rightsquigarrow AS] \rangle. \text{assembleOrders}()$,
- $\delta_4 = \textbf{pick } \langle \text{true}, \neg ([PP \rightsquigarrow RE] \vee [S \rightsquigarrow PS]) \rangle. \text{checkAssembledOrders}() \mid$
 pick $\langle \text{true}, [PP \rightsquigarrow RE] \vee [S \rightsquigarrow PS] \rangle. \text{outsourcingQualityCheck}()$,
- $\delta_5 = \textbf{pick } \langle \text{true}, [PP \rightsquigarrow AP] \wedge [S \rightsquigarrow AS] \rangle. \text{deliverOrder}()$.

Note that the program above is similar to the program in Example 4.3. The intuition of the program is also similar, except that here the program is decorated with context expressions that act as additional guards for each action invocation. The most different one is δ_4 . In δ_4 , depending on the context, we have a choice whether the quality check will be performed by the company or by outsourcing it. In particular, within the *peak season* ($[S \rightsquigarrow PS]$) or when the company wants to promote *resource efficiency* ($[PP \rightsquigarrow RE]$), the company outsource the quality control task. Additionally, note that we also need to consider the value domain theory. For instance, when the season is *winter holiday* (i.e., $[S \rightsquigarrow WH]$) together with the value domain theory it will implies that the season is peak season ($[S \rightsquigarrow PS]$), and hence the company will also outsource

the quality control task. Similarly, $[PP \rightsquigarrow WE]$ and $[PP \rightsquigarrow ME]$ will imply $[PP \rightsquigarrow RE]$. On the other hand, when neither $[S \rightsquigarrow PS]$ nor $[PP \rightsquigarrow RE]$ are implied by the current context together with the value domain theory, the company will perform the quality control by themselves.

With a slightly abuse of notation, below we provide another way to write the program above by also making use curly braces (“{”, “}”) for marking the scope of program operators:

```

while  $\exists x.[Order(x)] \wedge \neg[DeliveredOrder(x)]$  do {
  if  $\neg[\exists x.ApprovedOrder(x)]$ 
    then {pick  $\langle ReceivedOrder(x), [PP \rightsquigarrow AP] \wedge [S \rightsquigarrow AS] \rangle$ .approveOrder( $x$ )}
    else { $\varepsilon$ };
  pick  $\langle true, [PP \rightsquigarrow AP] \wedge [S \rightsquigarrow AS] \rangle$ .prepareOrders();
  pick  $\langle true, [PP \rightsquigarrow AP] \wedge [S \rightsquigarrow AS] \rangle$ .assembleOrders();
  { pick  $\langle true, \neg([PP \rightsquigarrow RE] \vee [S \rightsquigarrow PS]) \rangle$ .checkAssembledOrders() |
    pick  $\langle true, [PP \rightsquigarrow RE] \vee [S \rightsquigarrow PS] \rangle$ .outsourceQualityCheck();
  }
  pick  $\langle true, [PP \rightsquigarrow AP] \wedge [S \rightsquigarrow AS] \rangle$ .deliverOrder()
}

```

As for the initial Context, we consider the following Context:

$$C_0 = \{[PP \rightsquigarrow N], [S \rightsquigarrow NS]\}.$$

which capture the context (situation) where the normal processing plan is applied and the season is normal.

Furthermore, we have the set Π_C of context-evolution rules containing the following rules:

1. $\langle true, [S \rightsquigarrow PS] \rangle \mapsto \{[S \rightsquigarrow NS]\}.$
2. $\langle true, [PP \rightsquigarrow N] \wedge [S \rightsquigarrow NS] \rangle \mapsto \{[PP \rightsquigarrow WE], [S \rightsquigarrow PS]\}.$
3. $\langle true, [PP \rightsquigarrow RE] \wedge [S \rightsquigarrow PS] \rangle \mapsto \{[PP \rightsquigarrow N], [S \rightsquigarrow NS]\}.$
4. $\langle true, [S \rightsquigarrow AS] \rangle \mapsto \{ \}.$
5. $\langle \exists x.[Order(x)] \wedge \neg[DeliveredOrder(x)], [PP \rightsquigarrow WE] \wedge [S \rightsquigarrow PS] \rangle \mapsto \{ \}.$

The intuition of each rule above is as follows:

1. The first rule models the transition from *peak season* ($[S \rightsquigarrow PS]$) to *normal season* ($[S \rightsquigarrow NS]$), independently from the data.
2. The second rule models the transition from the situation where the processing plan is normal ($[PP \rightsquigarrow N]$) and the season is also normal ($[S \rightsquigarrow NS]$) into the situation where the season is peak ($[S \rightsquigarrow PS]$) and the worker efficiency ($[PP \rightsquigarrow WE]$) processing plan is applied.
3. The third rule models the transition from the situation where the company promotes resource efficiency ($[PP \rightsquigarrow RE]$) and the season is peak ($[S \rightsquigarrow PS]$) into the situation where the season is normal ($[S \rightsquigarrow NS]$) and the normal processing plan is applied ($[PP \rightsquigarrow N]$).
4. The fourth rule represents the transition where the context stay the same, independently from the current data and context. This is the case because the right

hand side of the rule is an empty set, and hence it does not change the assignment of any context dimensions. Additionally, the context expression $[S \leadsto AS]$ will be entailed no matter which value is assigned to the context dimension S .

5. The fifth rule represents the context-evolution where the context stay the same, given that (i) the current processing plan is worker efficiency, (ii) the current season is peak season, and (iii) there exists an order that is not yet delivered.

6.4.1 CSGKABs Standard Execution Semantics

As before, we are interested in verifying temporal properties over the evolution of CSGKABs, in particular “robust” properties that the system is required to guarantee independently from context changes. Towards this goal, we define the execution semantics of CSGKABs in terms of a possibly infinite-state transition system that simultaneously captures all possible evolutions of the system as well as all possible context changes.

Each state in the transition system of a CSGKAB is a tuple $\langle A, m, C, \delta \rangle$, where A is an ABox maintaining the current data, m is a service call map accounting for the service call results obtained so far, C is the current context, and δ is a program. The context selects which are the axioms of the contextual TBox that currently hold, in turn determining the current KB. Specifically, we introduce the notion of context-sensitive transition system in order to provide the semantics of CSGKAB as follows:

*Context-Sensitive
Transition System*

Definition 6.15 (Context-Sensitive Transition System). A *context-sensitive transition system* is a tuple $\mathcal{T}_{\mathcal{G}_{cx}} = \langle \Delta, T_{cx}, \Sigma, s_0, abox, ctx, \Rightarrow \rangle$, where:

1. T_{cx} is a contextualized TBox;
2. Σ is a set of states;
3. $s_0 \in \Sigma$ is the initial state;
4. $abox$ is a function that, given a state $s \in \Sigma$, returns the ABox associated to s ;
5. ctx is a function that, given a state $s \in \Sigma$, returns the context associated to s ;
6. $\Rightarrow \subseteq \Sigma \times \Sigma$ is a transition relation between pairs of states.

■

Starting from the initial state s_0 , $\mathcal{T}_{\mathcal{G}_{cx}}$ accounts for all the possible (simultaneous) data and context transitions. Technically, we revise the notion of executability of action invocation for GKABs by taking into account context expressions as follows: Let \mathcal{G}_{cx} be a CSGKAB, given a context-sensitive action invocation **pick** $\langle Q(\vec{p}), \varphi_C \rangle. \alpha(\vec{p})$, we say that a substitution σ , which substitutes the parameters \vec{p} with constants in Δ , is a *legal parameter assignment for α in A w.r.t. context C and action invocation **pick** $\langle Q(\vec{p}), \varphi_C \rangle. \alpha(\vec{p})$* if $\text{ASK}(Q\sigma, T_{cx}^C, A)$ is true.

To capture all possible context changes in a certain state, we define the notion of CTX-CHG relation that essentially captures all possible changes of a context based on the current ABox, the current context and the available context-evolution rules as follows.

Definition 6.16 (Context Change Relation CTX-CHG). Given a CSGKAB $\mathcal{G}_{cx} = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$, we define CTX-CHG relation of \mathcal{G}_{cx} such that given an ABox A , two contexts C and C' , we have a tuple $\langle A, C, C' \rangle \in \text{CTX-CHG}$ if there exists a context-evolution rule $\langle Q, \varphi_C \rangle \mapsto C_{new}$ in Π_C s.t.:

Context Change Relation

1. $\text{ASK}(Q, T_{cx}^C, A)$ is true;
2. $C \cup \Phi_D \models \varphi_C$;
3. for every context dimension $d \in \mathbb{D}$ s.t. $[d \rightsquigarrow v] \in C_{new}$, we have $[d \rightsquigarrow v] \in C'$;
4. for every context dimension $d \in \mathbb{D}$ s.t. $[d \rightsquigarrow v] \in C$, and there does not exist any v_2 s.t. $[d \rightsquigarrow v_2] \in C_{new}$, we have $[d \rightsquigarrow v] \in C'$.

■

Example 6.17. Consider our running example. We have

$$\langle A, \{[PP \rightsquigarrow N], [S \rightsquigarrow NS]\}, \{[PP \rightsquigarrow WE], [S \rightsquigarrow PS]\} \rangle \in \text{CTX-CHG}$$

where A is any ABox. In this case, the context-evolution rule that changes the values of the context dimensions is

$$\langle \text{true}, [PP \rightsquigarrow N] \wedge [S \rightsquigarrow NS] \rangle \mapsto \{[PP \rightsquigarrow WE], [S \rightsquigarrow PS]\}$$

As another example, we have

$$\langle A, \{[PP \rightsquigarrow N], [S \rightsquigarrow WH]\}, \{[PP \rightsquigarrow N], [S \rightsquigarrow NS]\} \rangle \in \text{CTX-CHG}$$

where A is any ABox. In this case, the context-evolution rule that changes the values of the context dimensions is

$$\langle \text{true}, [S \rightsquigarrow PS] \rangle \mapsto \{[S \rightsquigarrow NS]\}$$

Notice that in this example, the reason why the rule above is applicable is because *winter holiday* entail *peak season*.

Next, in order to take into account the presence of the context, we need to redefine the notion of (i) filter relation, (ii) TELL operation, (iii) final states, and (iv) program execution relation. For the refinement of the filter relation, because the TBox changes along with the context evolution, and also because sometimes we need the TBox w.r.t. the current context in order to construct the new ABox, thus we need to incorporate the corresponding context information inside the filter relation. Therefore, we then refine the filter relation as follows:

Definition 6.18 (Context-sensitive Filter Relation). A *Context-sensitive Filter Relation* f^{cx} is a relation that consists of tuples of the form $\langle A, F^+, F^-, C, A' \rangle$ such that $\emptyset \subseteq A' \subseteq ((A \setminus F^-) \cup F^+)$, where A and A' are ABoxes, C is a context, and F^+ as well as F^- are two sets of ABox assertions.

Context-sensitive Filter Relation f^{cx}

■

Roughly speaking, the filter relation indicates that the new ABox A' is constructed based on the current ABox A , the set of assertions to be added/deleted F^+/F^- and also the context C . The context C in f^{cx} will be mainly used later when we

incorporate the inconsistency handling mechanism that based on repair. Essentially, we need the context C during the repair in order to determine in which context we should do the repair and hence also determine which TBox assertions that we need to use. From now on, unless explicitly stated, for brevity, we simply say filter to refer to context-sensitive filter relation.

The TELL operation is then refined into CS-TELL operation as follows.

Context-sensitive
CS-TELL

Definition 6.19 (Context-sensitive CS-TELL Operation). Given a CSGKAB $\mathcal{G}_{cx} = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$ and a filter f^{cx} , we define CS-TELL $_{f^{cx}}$ as a relation such that we have a tuple $\langle \langle A, m, C \rangle, \alpha\sigma, \langle A', m', C' \rangle \rangle \in \text{CS-TELL}_{f^{cx}}$ if

1. σ is a legal parameter assignment for α in A w.r.t. context C and a certain action invocation **pick** $\langle Q(\vec{p}), \varphi_C \rangle . \alpha(\vec{p})$,
2. $\langle A, C, C' \rangle \in \text{CTX-CHG}$,
3. there exists $\theta \in \text{EVAL}(\text{ADD}(T_{cx}^C, A, \alpha\sigma))$ such that:
 - a) for each skolem term $f(c) \in \text{DOM}(m) \cap \text{DOM}(\theta)$, we have $f(c)/v \in m$ if and only if $f(c)/v \in \theta$ (i.e., θ and m “agree” on the common skolem terms in their domains, in order to realize the deterministic service call semantics);
 - b) $m' = m \cup \theta$;
 - c) $\langle A, \text{ADD}(T_{cx}^C, A, \alpha\sigma)\theta, \text{DEL}(T_{cx}^C, A, \alpha\sigma), C', A' \rangle \in f^{cx}$;
 - d) A is T_{cx}^C -consistent, and A' is $T_{cx}^{C'}$ -consistent.

■

Essentially, instead of only capturing the changes of ABox and service call map by an action, the CS-TELL operation also capture the changes of the contexts. In addition, the inconsistency check is performed with respect to the new context. Next, we refine the notion of final states as follows.

Final State

Definition 6.20 (Final State). Let $\mathcal{G}_{cx} = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$ be a CSGKAB with transition system $\mathcal{T}_{\mathcal{G}_{cx}}$. We define when a state $\langle A, m, C, \delta \rangle$ of $\mathcal{T}_{\mathcal{G}_{cx}}$ is a *final state*, written $\langle A, m, C, \delta \rangle \in \mathbb{F}$, as follows:

1. $\langle A, m, C, \varepsilon \rangle \in \mathbb{F}$;
2. $\langle A, m, C, \delta_1 | \delta_2 \rangle \in \mathbb{F}$ if $\langle A, m, C, \delta_1 \rangle \in \mathbb{F}$ or $\langle A, m, C, \delta_2 \rangle \in \mathbb{F}$;
3. $\langle A, m, C, \delta_1 ; \delta_2 \rangle \in \mathbb{F}$ if $\langle A, m, C, \delta_1 \rangle \in \mathbb{F}$ and $\langle A, m, C, \delta_2 \rangle \in \mathbb{F}$;
4. $\langle A, m, C, \text{if } \varphi \text{ then } \delta_1 \text{ else } \delta_2 \rangle \in \mathbb{F}$
if $\text{ASK}(\varphi, T, A) = \text{true}$, and $\langle A, m, C, \delta_1 \rangle \in \mathbb{F}$;
5. $\langle A, m, C, \text{if } \varphi \text{ then } \delta_1 \text{ else } \delta_2 \rangle \in \mathbb{F}$
if $\text{ASK}(\varphi, T, A) = \text{false}$, and $\langle A, m, C, \delta_2 \rangle \in \mathbb{F}$;
6. $\langle A, m, C, \text{while } \varphi \text{ do } \delta \rangle \in \mathbb{F}$ if $\text{ASK}(\varphi, T, A) = \text{false}$;
7. $\langle A, m, C, \text{while } \varphi \text{ do } \delta \rangle \in \mathbb{F}$ if $\text{ASK}(\varphi, T, A) = \text{true}$, and $\langle A, m, C, \delta \rangle \in \mathbb{F}$.

■

Having the refined filter relation, CS-TELL operation, and also the refined final states, we now proceed to refine the *program execution relation* $\xrightarrow{\alpha\sigma, f^{cx}}$, which describes how a grounded action simultaneously evolves the contexts as well as data- and program-state.

Definition 6.21 (Context-sensitive Program Execution Relation). Given a CSGKAB $\mathcal{G}_{cx} = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$, and a filter relation f^{cx} , we define a *context-sensitive program execution relation* $\xrightarrow{\alpha\sigma, f^{cx}}$ as follows:

Context-sensitive
Program Execution
Relation

1. $\langle A, m, C, \mathbf{pick} \langle Q(\vec{p}), \varphi_C \rangle . \alpha(\vec{p}) \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A', m', C', \varepsilon \rangle$,
if the following hold:
 - a) $\langle \langle A, m, C \rangle, \alpha\sigma, \langle A', m', C' \rangle \rangle \in \text{CS-TELL}_{f^{cx}}$,
 - b) σ is a legal parameter assignment for α in A w.r.t. context C and action invocation $\mathbf{pick} \langle Q(\vec{p}), \varphi_C \rangle . \alpha(\vec{p})$,
 - c) $C \cup \Phi_D \models \varphi_C$.
2. $\langle A, m, C, \delta_1 | \delta_2 \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A', m', C', \delta' \rangle$,
if $\langle A, m, C, \delta_1 \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A', m', C', \delta' \rangle$ or $\langle A, m, \delta_2 \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A', m', C', \delta' \rangle$;
3. $\langle A, m, C, \delta_1; \delta_2 \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A', m', C', \delta'_1; \delta_2 \rangle$,
if $\langle A, m, C, \delta_1 \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A', m', C', \delta'_1 \rangle$;
4. $\langle A, m, C, \delta_1; \delta_2 \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A', m', C', \delta'_2 \rangle$,
if $\langle A, m, C, \delta_1 \rangle \in \mathbb{F}$, and $\langle A, m, C, \delta_2 \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A', m', C', \delta'_2 \rangle$;
5. $\langle A, m, C, \mathbf{if} \varphi \mathbf{then} \delta_1 \mathbf{else} \delta_2 \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A', m', C', \delta'_1 \rangle$,
if $\text{ASK}(\varphi, T, A) = \text{true}$, and $\langle A, m, C, \delta_1 \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A', m', C', \delta'_1 \rangle$;
6. $\langle A, m, C, \mathbf{if} \varphi \mathbf{then} \delta_1 \mathbf{else} \delta_2 \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A', m', C', \delta'_2 \rangle$,
if $\text{ASK}(\varphi, T, A) = \text{false}$, and $\langle A, m, C, \delta_2 \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A', m', C', \delta'_2 \rangle$;
7. $\langle A, m, C, \mathbf{while} \varphi \mathbf{do} \delta \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A', m', C', \delta'; \mathbf{while} \varphi \mathbf{do} \delta \rangle$,
if $\text{ASK}(\varphi, T, A) = \text{true}$, and $\langle A, m, C, \delta \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A', m', C', \delta' \rangle$.

■

We are now defining the construction of CSGKABs transition systems that is parameterized with filter as follows.

Definition 6.22 (CSGKAB Transition System). Given a CSGKAB $\mathcal{G}_{cx} = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$, and a filter relation f^{cx} , we define the *transition system of \mathcal{G}_{cx} w.r.t. f^{cx}* , written $\mathcal{T}_{\mathcal{G}_{cx}}^{f^{cx}}$, as $\langle \Delta, T_{cx}, \Sigma, s_0, \text{abox}, \text{ctx}, \Rightarrow \rangle$, where

CSGKAB Transition
System

1. $s_0 = \langle A_0, \emptyset, C_0, \delta \rangle$, and
2. Σ and \Rightarrow are defined by simultaneous induction as the smallest sets such that
 - a) $s_0 \in \Sigma$, and
 - b) if $\langle A, m, C, \delta \rangle \in \Sigma$ and $\langle A, m, C, \delta \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A', m', C', \delta' \rangle$, then $\langle A', m', C', \delta' \rangle \in \Sigma$ and $\langle A, m, C, \delta \rangle \Rightarrow \langle A', m', C', \delta' \rangle$.

■

As in GKABs, by suitably concretizing the filter relation, we can obtain various execution semantics for CSGKABs. We are now exploiting filter relations to define the standard execution semantics of CSGKAB. Particularly, we define a context-sensitive standard filter relation f_S^{cx} as follows:

Definition 6.23 (Context-sensitive Standard Filter f_S^{cx}). Let $\mathcal{G}_{cx} = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$ be a CSGKAB, A and A' be ABoxes over $\text{voc}(T)$, F^+ be a set of ABox assertions over $\text{voc}(T)$ to be added, F^- be a set of ABox assertions over $\text{voc}(T)$ to be deleted, and C be a context, we then have $\langle A, F^+, F^-, C, A' \rangle \in f_S^{cx}$ if $A' = (A \setminus F^-) \cup F^+$,

Context-sensitive
Standard Filter f_S^{cx}

■

Filter f_S^{cx} gives rise to the *standard execution semantics* for \mathcal{G}_{cx} . We call the CSGKABs adopting these semantics *S-CSGKABs*.

*S-CSGKABs
Standard Transition
System*

Definition 6.24 (S-CSGKABs Standard Transition System). Given a CSGKAB $\mathcal{G}_{cx} = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$ and a standard filter f_S^{cx} , the *standard transition system* of \mathcal{G}_{cx} , written $\mathcal{T}_{\mathcal{G}_{cx}}^{f_S^{cx}}$, is the transition system of \mathcal{G}_{cx} w.r.t. f_S^{cx} . ■

The notion of run and run-boundedness of S-CSGKABs transition systems is defined similarly as in Definitions 3.27 and 3.28.

Example 6.25. Continuing our running example, let the CSGKAB \mathcal{G}_{cx} specified in Example 6.14 be an S-CSGKAB. Consider a state $s = \langle A, m, C, \delta \rangle$ where:

- $A = \{ \text{ReceivedOrder}(\text{chair}), \text{ApprovedOrder}(\text{table}), \text{designedBy}(\text{table}, \text{alice}), \text{Designer}(\text{alice}), \text{hasDesign}(\text{table}, \text{ecodesign}), \text{hasAssemblingLoc}(\text{table}, \text{bolzano}) \},$
- $m = \{ [\text{GETDESIGNER}(\text{table}) \rightarrow \text{alice}], [\text{GETDESIGN}(\text{table}) \rightarrow \text{ecodesign}], [\text{ASSIGNASSEMBLINGLOC}(\text{table}) \rightarrow \text{bolzano}] \},$
- $C = \{ [\text{PP} \rightsquigarrow \text{N}], [\text{S} \rightsquigarrow \text{NS}] \},$
- $\delta = \delta_3; \delta_4; \delta_5; \text{while } \exists x. [\text{Order}(x)] \wedge \neg [\text{DeliveredOrder}(x)] \text{ do } \delta_0.$

Observe that the state s is a reachable state from the initial state s_0 in the transition system $\mathcal{T}_{\mathcal{G}_{cx}}^{f_S^{cx}}$ of \mathcal{G}_{cx} . From the state s , we have a possible successor state $s' = \langle A', m', C', \delta' \rangle$ with

- $A' = \{ \text{ReceivedOrder}(\text{chair}), \text{designedBy}(\text{table}, \text{alice}), \text{Designer}(\text{alice}), \text{hasDesign}(\text{table}, \text{ecodesign}), \text{hasAssemblingLoc}(\text{table}, \text{bolzano}), \text{AssembledOrder}(\text{table}), \text{assembledBy}(\text{table}, \text{bob}), \text{Assembler}(\text{bob}) \},$
- $m' = \{ [\text{GETDESIGNER}(\text{table}) \rightarrow \text{alice}], [\text{GETDESIGN}(\text{table}) \rightarrow \text{ecodesign}], [\text{ASSIGNASSEMBLINGLOC}(\text{table}) \rightarrow \text{bolzano}], [\text{GETASSEMBLER}(\text{table}) \rightarrow \text{bob}], [\text{GETASSEMBLINGLOC}(\text{table}) \rightarrow \text{bolzano}] \},$
- $C' = \{ [\text{PP} \rightsquigarrow \text{WE}], [\text{S} \rightsquigarrow \text{PS}] \},$
- $\delta' = \delta_4; \delta_5; \text{while } \exists x. [\text{Order}(x)] \wedge \neg [\text{DeliveredOrder}(x)] \text{ do } \delta_0.$

The state s' is obtained from the execution of action invocation

pick $\langle \text{true}, [\text{PP} \rightsquigarrow \text{AP}] \wedge [\text{S} \rightsquigarrow \text{AS}] \rangle. \text{assembleOrders}()$

where the context is changing from C to C' due to the application of the following context evolution rule:

$$\langle \text{true}, [\text{PP} \rightsquigarrow \text{N}] \wedge [\text{S} \rightsquigarrow \text{NS}] \rangle \mapsto \{[\text{PP} \rightsquigarrow \text{WE}], [\text{S} \rightsquigarrow \text{PS}]\}.$$

Next, in δ_4 there are choices between doing the quality check internally or by outsourcing it.

$$\begin{aligned} \delta_4 = & \text{pick } \langle \text{true}, \neg([\text{PP} \rightsquigarrow \text{RE}] \vee [\text{S} \rightsquigarrow \text{PS}]) \rangle . \text{checkAssembledOrders}() \mid \\ & \text{pick } \langle \text{true}, [\text{PP} \rightsquigarrow \text{RE}] \vee [\text{S} \rightsquigarrow \text{PS}] \rangle . \text{outsourceQualityCheck}() \end{aligned}$$

Since the current context is C' , and $C' \cup \Phi_{\text{D}} \models [\text{PP} \rightsquigarrow \text{RE}] \vee [\text{S} \rightsquigarrow \text{PS}]$, here we can execute `outsourceQualityCheck/0`. In this case one plausible successor of s' is $s'' = \langle A'', m'', C'', \delta'' \rangle$ with

- $A'' = \{ \text{ReceivedOrder}(\text{chair}), \text{designedBy}(\text{table}, \text{alice}), \text{Designer}(\text{alice}), \text{hasDesign}(\text{table}, \text{ecodesign}), \text{hasAssemblingLoc}(\text{table}, \text{bolzano}), \text{AssembledOrder}(\text{table}), \text{assembledBy}(\text{table}, \text{bob}), \text{Assembler}(\text{bob}), \text{checkedBy}(\text{table}, \text{qccompany}) \}$,
- $m'' = \{ [\text{GETDESIGNER}(\text{table}) \rightarrow \text{alice}], [\text{GETDESIGN}(\text{table}) \rightarrow \text{ecodesign}], [\text{ASSIGNASSEMBLINGLOC}(\text{table}) \rightarrow \text{bolzano}], [\text{GETASSEMBLER}(\text{table}) \rightarrow \text{bob}], [\text{GETASSEMBLINGLOC}(\text{table}) \rightarrow \text{bolzano}], [\text{GETQCSERVICE}(\text{table}) \rightarrow \text{qccompany}] \}$,
- $C'' = \{[\text{PP} \rightsquigarrow \text{WE}], [\text{S} \rightsquigarrow \text{PS}]\}$,
- $\delta'' = \delta_5$; **while** $\exists x. [\text{Order}(x)] \wedge \neg [\text{DeliveredOrder}(x)]$ **do** δ_0 .

In this case, the value of each context dimension stay the same due to the application of the following context evolution rule: $\langle \text{true}, [\text{S} \rightsquigarrow \text{AS}] \rangle \mapsto \{\}$.

Rejecting Inconsistent State in S-CSGKABS. Recall our state s above, one of its successor state is the state s' . Now, consider a state $s''' = \langle A''', m''', C''', \delta''' \rangle$ with $A''' = A'$, $m''' = m'$, $C''' = \{[\text{PP} \rightsquigarrow \text{N}], [\text{S} \rightsquigarrow \text{NS}]\}$, and $\delta''' = \delta'$. In this case, similar to s' , the execution of on the state s

$$\text{pick } \langle \text{true}, [\text{PP} \rightsquigarrow \text{AP}] \wedge [\text{S} \rightsquigarrow \text{AS}] \rangle . \text{assembleOrders}()$$

might leads us into s''' where the value of each context dimension stay the same due to the application of the following context evolution rule:

$$\langle \text{true}, [\text{S} \rightsquigarrow \text{AS}] \rangle \mapsto \{\}.$$

Though s' and s''' only differ on their contexts, we have that s' is consistent while s''' is inconsistent due to the following TBox assertions:

$$\langle \text{Designer} \sqsubseteq \neg \text{Assembler} : [\text{PP} \rightsquigarrow \text{N}] \wedge ([\text{S} \rightsquigarrow \text{NS}] \vee [\text{S} \rightsquigarrow \text{LS}]) \rangle$$

$$\langle \text{Designer} \sqsubseteq \text{Assembler} : [\text{PP} \leadsto \text{WE}] \vee [\text{S} \leadsto \text{PS}] \rangle.$$

From this example, we have seen that depending on the context, some domain constraint might be activated or not. Thus, the context change influence the consistency of a state. In the case the construction of transition system $\mathcal{T}_{\mathcal{G}_{cx}}^{f_{S^{cx}}}$ will reject the generation of s''' .

6.5 Verifying Temporal Properties over Standard CSGKAB

We are now interested in verifying whether the evolution of an S-CSGKAB \mathcal{G}_{cx} , which is represented by $\mathcal{T}_{\mathcal{G}_{cx}}$, complies with some temporal properties.

6.5.1 Context-Sensitive FO-Variant of μ -calculus

As previous, in order to specify temporal properties to be verified over S-CSGKABs, we use a first-order variant of μ -calculus [170, 149]. In particular, we introduce the language $\mu\mathcal{L}_{\text{CTX}}$ of *context-sensitive temporal properties*, which is based on $\mu\mathcal{L}_A^{\text{EQL}}$ (see Section 2.6.1). Basically, we exploit ECQs to query the states, and support a first-order quantification across states, where the quantification ranges over the constants in the current active domain. Additionally, we augment $\mu\mathcal{L}_{\text{CTX}}$ with context expressions, which allows us to check also context information while querying states. Formally,

Syntax of $\mu\mathcal{L}_{\text{CTX}}$ $\mu\mathcal{L}_{\text{CTX}}$ is defined as follows :

$$\Phi := Q \mid \varphi_C \mid \neg\Phi \mid \Phi_1 \vee \Phi_2 \mid \exists x.\Phi \mid \langle \neg \rangle \Phi \mid Z \mid \mu Z.\Phi$$

where φ_C is a context expression over \mathcal{L}_{cx} and the rest are the same as in $\mu\mathcal{L}_A^{\text{EQL}}$ (see Section 2.6.1). Similar to $\mu\mathcal{L}_A^{\text{EQL}}$, let $\langle T_{cx}, A_0 \rangle$ be a contextualized KB, we call a $\mu\mathcal{L}_{\text{CTX}}$ formula Φ *is over* $\langle T_{cx}, A_0 \rangle$ if each query Q in Φ is a query over $\langle T_{cx}, A_0 \rangle$ (i.e., each atom in Q only use the vocabulary from $\text{VOC}(T_{cx})$ and might uses constants in $\text{ADOM}(A_0)$).

Semantics of $\mu\mathcal{L}_{\text{CTX}}$

The semantics of $\mu\mathcal{L}_{\text{CTX}}$ is also defined over a (possibly infinite) transition system $\mathcal{T} = \langle \Delta, T_{cx}, \Sigma, s_0, abox, ctx, \Rightarrow \rangle$. Similar to $\mu\mathcal{L}_A^{\text{EQL}}$, given a transition system \mathcal{T} , in order to assign the meaning to $\mu\mathcal{L}_{\text{CTX}}$ formulas, we introduce an individual variable valuation v , i.e., a mapping from individual variables x to Δ , and a predicate variable valuation V , i.e., a mapping from predicate variables Z to subsets of Σ . The semantics of $\mu\mathcal{L}_{\text{CTX}}$ follows the standard μ -calculus semantics, except for the semantics of queries and of quantification. We assign meaning to $\mu\mathcal{L}_{\text{CTX}}$ formulas by associating to \mathcal{T} , v

and V an *extension function* $(\cdot)_{v,V}^{\mathcal{T}}$, which maps $\mu\mathcal{L}_{\text{CTX}}$ formulas to subsets of Σ . The extension function $(\cdot)_{v,V}^{\mathcal{T}}$ is defined inductively as follows:

$$\begin{aligned}
(Q)_{v,V}^{\mathcal{T}} &= \{s \in \Sigma \mid \text{CERT}(Qv, T_{cx}^{ctx(s)}, abox(s)) = \text{true}\} \\
(\varphi_C)_{v,V}^{\mathcal{T}} &= \{s \in \Sigma \mid ctx(s) \cup \Phi_{\mathbb{D}} \models \varphi_C\} \\
(\exists x.\Phi)_{v,V}^{\mathcal{T}} &= \{s \in \Sigma \mid \exists d.d \in \text{ADOM}(abox(s)) \text{ and } s \in (\Phi)_{v[x/d],V}^{\mathcal{T}}\} \\
(Z)_{v,V}^{\mathcal{T}} &= V(Z) \subseteq \Sigma \\
(\neg\Phi)_{v,V}^{\mathcal{T}} &= \Sigma - (\Phi)_{v,V}^{\mathcal{T}} \\
(\Phi_1 \vee \Phi_2)_{v,V}^{\mathcal{T}} &= (\Phi_1)_{v,V}^{\mathcal{T}} \cup (\Phi_2)_{v,V}^{\mathcal{T}} \\
(\langle \neg \rangle \Phi)_{v,V}^{\mathcal{T}} &= \{s \in \Sigma \mid \exists s'. s \Rightarrow s' \text{ and } s' \in (\Phi)_{v,V}^{\mathcal{T}}\} \\
(\mu Z.\Phi)_{v,V}^{\mathcal{T}} &= \bigcap \{\mathcal{E} \subseteq \Sigma \mid (\Phi)_{v,V[Z/\mathcal{E}]}^{\mathcal{T}} \subseteq \mathcal{E}\}
\end{aligned}$$

where Qv is the query obtained from Q by substituting its free variables according to v . For a closed formula Φ (for which $(\Phi)_{v,V}^{\mathcal{T}}$ does not depend on v or V), we denote with $(\Phi)^{\mathcal{T}}$ the extension of Φ in \mathcal{T} , and we say that Φ holds in a state $s \in \Sigma$ if $s \in (\Phi)^{\mathcal{T}}$. In this case, we write $\mathcal{T}, s \models \Phi$. Furthermore, a closed formula Φ holds in \mathcal{T} , briefly \mathcal{T} *satisfies* Φ , if $\mathcal{T}, s_0 \models \Phi$ (In this situation we write $\mathcal{T} \models \Phi$).

Example 6.26. In our running example, the property

$$\nu Z.(\forall x.\text{Order}(x) \wedge [S \leadsto \text{PS}] \rightarrow \mu Y.(\text{DeliveredOrder}(x) \vee \langle \neg \rangle Y)) \wedge [\neg]Z$$

checks that along every path, it is always true that every customer order in the peak season will be eventually delivered, independently on how the context and the state evolve.

6.5.2 Verification of Standard CSGKABs

The problem definition of the $\mu\mathcal{L}_{\text{CTX}}$ formula verification over S-CSGKABs is defined similarly as in KABs (see Definition 3.13). Precisely it is defined as follows:

Definition 6.27 (Verification of a $\mu\mathcal{L}_{\text{CTX}}$ Property over an S-CSGKAB). Given an S-CSGKAB $\mathcal{G}_{cx} = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$ and a closed $\mu\mathcal{L}_{\text{CTX}}$ formula Φ over $\langle T_{cx}, A_0 \rangle$. Let $\mathcal{T}_{\mathcal{G}_{cx}}$ be the transition system of \mathcal{G}_{cx} , the *verification of a $\mu\mathcal{L}_{\text{CTX}}$ formula Φ over \mathcal{G}_{cx}* is a problem to check whether $\mathcal{T}_{\mathcal{G}_{cx}} \models \Phi$. ■

Verification of a $\mu\mathcal{L}_{\text{CTX}}$ Formula over an S-CSGKAB

We solve this problem by compiling S-CSGKABs into S-GKABs and show that the verification of $\mu\mathcal{L}_{\text{CTX}}$ formulas over S-CSGKABs can be recast as verification over S-GKAB (This claim is formally stated in Theorem 6.50). Technically, we do the following:

1. We define a generic translation τ_{cx} (in Section 6.5.2.1), that given an S-CSGKABs \mathcal{G}_{cx} , produces an S-GKAB $\tau_{cx}(\mathcal{G}_{cx})$.
2. We define a generic translation t_{cx} (in Section 6.5.2.1) that takes a $\mu\mathcal{L}_{\text{CTX}}$ formula Φ as an input and produces a $\mu\mathcal{L}_A^{\text{EQL}}$ formula $t_{cx}(\Phi)$.
3. We introduce a certain bisimulation relation in which given a context-sensitive transition system \mathcal{T}_1 and a KB transition system \mathcal{T}_2 such that they are bisimilar

w.r.t. this bisimulation relation, we have that \mathcal{T}_1 satisfy a $\mu\mathcal{L}_{\text{CTX}}$ formula Φ if and only if \mathcal{T}_2 satisfy the $\mu\mathcal{L}_A^{\text{EQL}}$ formula $t_{cx}(\Phi)$.

4. We show that the transition system of an S-CSGKAB \mathcal{G}_{cx} and the transition system of the corresponding S-GKAB $\tau_{cx}(\mathcal{G}_{cx})$ (that is obtained from \mathcal{G}_{cx} via the translation τ_{cx}) are bisimilar w.r.t. the bisimulation relation introduced in the previous step.

6.5.2.1 Transforming S-CSGKABs into S-GKABs

Recall that we fix a set

$$\mathbb{D} = \{d_1, \dots, d_n\}$$

of context dimensions. Each context dimension $d_i \in \mathbb{D}$ has its own tree-shaped finite value domain $\langle \text{Dom}(d_i), \prec_{d_i} \rangle$, where $\text{Dom}(d_i)$ represents the finite set of domain values, and \prec_{d_i} represents the predecessor relation forming the tree.

The idea of our translation τ_{cx} , that translates S-CSGKABs \mathcal{G}_{cx} into an S-GKAB $\tau_{cx}(\mathcal{G}_{cx})$, is as follows:

1. We capture the context information using ABox assertions. Precisely, each context dimension assignment is internally captured by an ABox assertion. Thus, for each context dimension assignment $[d_i \rightsquigarrow v_j]$ we reserve two fresh concept names $D_i^{v_j}$ and $\mathbf{D}_i^{v_j}$ in order to represent it as an ABox assertion. Such kind of concept name is called *context dimension concept name*. The reason for introducing two different concepts for representing a context dimension assignment is to simplify the correctness proof of the reduction. The idea is to make a separation between the following situations:
 - a) when we need to reason using only the current context information (in this case we use $\mathbf{D}_i^{v_j}$).
 - b) when we need to use the current context information together with the value domain semantics of each context dimension value domain (in this case we use $D_i^{v_j}$).

Furthermore, we reserve a special constant $\mathbf{c} \in \Delta_0$ to populate such kind of concept name. We call *context ABox assertion* an ABox assertion made by context dimension concepts. Furthermore, the semantics of context dimensions value domains is captured inside the TBox.

2. The context expressions are captured by ECQs queries which use context dimension concepts as its vocabulary.
3. We simulate the context-evolution rules by actions.
4. To check the inconsistency, since the TBox assertions that is needed to check inconsistency are determined based on the context, we introduce several special actions to check inconsistency which also taking into account the context. To this aim, for a technical reason, we reserve a fresh concept name Inc and we will use the TBox assertion $Inc \sqsubseteq \neg Inc$ to prevent the generation of an inconsistent state, and we also use \mathbf{c} to populate such concept.
5. For translating the program, the idea is to concatenate each action execution with non-deterministic choice of actions that change context and the action that checks the inconsistency. Furthermore, since the changes of context requires the original ABox, we don't materialize the result of an action execution directly after its execution, instead we just mark the assertions that should

Context Dimension
Concept Name

Context ABox
Assertion

be added/deleted, and materialized it during the execution of the action that evolves context. To this aim, for each concept name $N \in \text{VOC}(T_{cx})$, we introduce two fresh concept name N^a and N^d to keep track the temporary information about ABox assertions to be added/deleted before we materialize the update (similarly for roles).

6. We also use a special marker $\text{State}(temp)$, made by a reserved concept name State and a constant $temp$, to mark intermediate states (the states where we still need to change the context and check the inconsistency). We call a stable state the state except the intermediate states (i.e., the states that does not contain $\text{State}(temp)$).

Now, in order to reduce the verification of $\mu\mathcal{L}_{\text{CTX}}$ over S-CSGKAB into the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ over S-GKAB, we first need to introduce several preliminaries below.

Definition 6.28 (Set of All Possible Contexts Over \mathbb{D}). We define *the set of all possible contexts over \mathbb{D}* as a set $\text{CTX}(\mathbb{D})$ of contexts such that $C \in \text{CTX}(\mathbb{D})$ if C is a context over \mathbb{D} . ■

Set of All Possible Contexts

Roughly speaking, the definition above stated that the set $\text{CTX}(\mathbb{D})$ contains all possible context over \mathbb{D} .

We now proceed to define a TBox which capture the semantics of context dimensions value domains in terms of TBox assertions. I.e., this TBox capture the value domain theory $\Phi_{\mathbb{D}}$ of \mathbb{D} .

Definition 6.29 (TBox obtained from a Set of Context Dimensions \mathbb{D}). We define a *TBox obtained from a set of context dimensions \mathbb{D}* as a $DL\text{-}Lite_A$ TBox $T_{\mathbb{D}}$ such that:

TBox obtained from a Set of Context Dimensions \mathbb{D}

- For each $d_i \in \mathbb{D}$, and for all values $v_1, v_2 \in \text{Dom}(d_i)$ such that $v_1 \prec_d v_2$, we have that $T_{\mathbb{D}}$ contains $D_i^{v_1} \sqsubseteq D_i^{v_2}$, where $D_i^{v_1}$ and $D_i^{v_2}$ are fresh concept names each representing the context dimension assignment $[d_i \rightsquigarrow v_1]$ and $[d_i \rightsquigarrow v_2]$ respectively. Intuitively, this states that the value v_2 is more general than v_1 , and hence, whenever we have $[d \rightsquigarrow v_1]$ we can infer that $[d \rightsquigarrow v_2]$.
- For each $d_i \in \mathbb{D}$, and for all values $v_1, v_2, v \in \text{Dom}(d_i)$ such that $v_1 \prec_d v$ and $v_2 \prec_d v$, we have that $T_{\mathbb{D}}$ contains $D_i^{v_1} \sqsubseteq \neg D_i^{v_2}$, where $D_i^{v_1}$ and $D_i^{v_2}$ are fresh concept names each representing the context dimension assignment $[d_i \rightsquigarrow v_1]$ and $[d_i \rightsquigarrow v_2]$ respectively. Intuitively, this expresses that sibling values v_1 and v_2 are disjoint. ■

Notice that in the definition above we use the context dimension concepts that are used to reason in the situation where we take into account the value domain semantics. Since the value domain semantics is encoded only using those concept names, it will be ignored when we make a query using the context dimension concepts that are used to reason without considering the value domain semantics.

We now define a query that represents a context as follows:

Definition 6.30 (A Query That Represents Context C). Given a context

A Query That Represents Context C

$$C = \{[d_1 \rightsquigarrow v_1], \dots, [d_n \rightsquigarrow v_m]\},$$

the *query that represents context C* is a boolean CQ

$$q_C = \mathbf{D}_1^{v_1}(\mathbf{c}) \wedge \dots \wedge \mathbf{D}_n^{v_m}(\mathbf{c})$$

■

Note that in the definition above we use the context dimension concepts that are used to reason without considering the value domain semantics. Later on, we will see that the query above can be used to check the current context.

To capture the context information within an ABox, we define the notion of a set of ABox assertions representing a context as follows:

*Set of ABox
Assertions
Representing a
Context*

Definition 6.31 (Set of ABox Assertions Representing a Context). Let

$$C = \{[d_1 \rightsquigarrow v_1], \dots, [d_n \rightsquigarrow v_m]\}$$

be a context, the *set of ABox assertions representing the context C* is a set A_C of context ABox assertions as follows:

$$A_C = \{D_1^{v_1}(\mathbf{c}), \dots, D_n^{v_m}(\mathbf{c}), \mathbf{D}_1^{v_1}(\mathbf{c}), \dots, \mathbf{D}_n^{v_m}(\mathbf{c})\}.$$

where each $D_i^{v_j}$ (resp. $\mathbf{D}_i^{v_j}$) is a context dimension concept name. ■

When we compile S-CSGKABs into S-GKABs, we represent a context expression as a query and it is done as follows:

*A Query that
Represents a Context
Expressions*

Definition 6.32 (A Query That Represents A Context Expressions). Given a context expressions φ_C , the *query that represents the context expressions φ_C* is a boolean ECQ query q_{φ_C} obtained by replacing each occurrence of context dimension assignment of the form $[d_i \rightsquigarrow v_j]$ with an atom $D_i^{v_j}(\mathbf{c})$, where $D_i^{v_j}$ (resp. \mathbf{c}) is the reserved concept name (resp. reserved constant) explained before. ■

Notice that in the definition above we use the context dimension concepts that are used to reason in the situation where we take into account the value domain semantics. In the following lemma we show the “correctness” of our mechanism in expressing a context expression as a query.

Lemma 6.33. *Given a context C, and a context expression φ_C . Let q_{φ_C} be the query that represents the context expressions φ_C . We have that*

$$C \cup \Phi_{\mathbb{D}} \models \varphi_C \text{ if and only if } \text{CERT}(q_{\varphi_C}, T_{\mathbb{D}}, A_C) = \text{true}$$

Proof. Trivially true by observing Definitions 6.29, 6.31 and 6.32. □

We now proceed to describe how we compile queries when we transform S-CSGKABs into S-GKABs such that they will be answered using the correct TBox w.r.t. the corresponding context.

*Contextually
Compiled Query*

Definition 6.34 (Contextually Compiled Query). Given an ECQ Q , a *contextually compiled query* of Q w.r.t. \mathbb{D} is a query Q_{cx} of the form

$$Q_{cx} = \left(\bigvee_{C \in \text{CTX}(\mathbb{D})} (q_C \wedge \text{rew}(Q, T_{cx}^C)) \right)$$

where q_C is the query obtained from the context C . ■

In the following lemma we show the “correctness” of our contextually compiled query.

Lemma 6.35. *Given a contextualized KB $\langle T_{cx}, A \rangle$, an ECQ Q over $\text{VOC}(T_{cx})$ and a context C over \mathbb{D} . Let Q_{cx} be the contextually compiled query of Q w.r.t. \mathbb{D} , A_C be the set of ABox assertions representing the context C , and $T_{\mathbb{D}}$ be the TBox obtained from \mathbb{D} . We have*

$$\text{CERT}(Q, T_{cx}^C, A) = \text{CERT}(Q_{cx}, T_{\mathbb{D}}, A \cup A_C)$$

Proof. Let $Q_{cx} = \left(\bigvee_{C \in \text{CTX}(\mathbb{D})} (q_C \wedge \text{rew}(Q, T_{cx}^C)) \right)$ where q_C is the query obtained from the context C . By Theorem 2.40, we have $\text{CERT}(Q, T_{cx}^C, A) = \text{ANS}(\text{rew}(Q, T_{cx}^C), A)$, and $\text{CERT}(Q_{cx}, T_{\mathbb{D}}, A \cup A_C) = \text{ANS}(\text{rew}(Q_{cx}, T_{\mathbb{D}}), A \cup A_C)$. Since Q_{cx} doesn't use any vocabulary from $\text{VOC}(T_{\mathbb{D}})$ then $\text{rew}(Q_{cx}, T_{\mathbb{D}}) = Q_{cx}$. Now we have to show that $\text{ANS}(\text{rew}(Q, T_{cx}^C), A) = \text{ANS}(Q_{cx}, A \cup A_C)$. By construction of Q_{cx} and q_C (see Definitions 6.30 and 6.34) we have that

1. there exists $C \in \text{CTX}(\mathbb{D})$ such that $\text{ANS}(q_C, A \cup A_C) = \text{true}$ and
2. for all $C' \in \text{CTX}(\mathbb{D})$ such that $C' \neq C$ we have $\text{ANS}(q_{C'}, A \cup A_C) = \text{false}$.

Thus, now we only need to show that

$$\text{ANS}(\text{rew}(Q, T_{cx}^C), A) = \text{ANS}(q_C \wedge \text{rew}(Q, T_{cx}^C), A \cup A_C).$$

The proof is then easily completed by observing that Q doesn't use any context dimension concepts, hence we can ignore A_C . \square

Recall that in S-GKABs we simulate the action execution and the context change in two different steps. Furthermore, since the context change is conducted after the action execution and it requires the original ABox, we can not immediately update the ABox after the action execution. Therefore, now we introduce the notion of delayed action that does not add/delete ABox assertions immediately, but only adds markers about which ABox assertions that should be added or deleted. Essentially when we compile S-CSGKABs into S-GKABs, we will see it later that we translate each action into a delayed action.

Definition 6.36 (Delayed Action). Given an action $\alpha(\vec{p}) : \{e_1, \dots, e_n\}$ with $e_i = [q_i^+] \wedge Q_i^- \rightsquigarrow \mathbf{add} F_i^+, \mathbf{del} F_i^-$, a *delayed action* obtained from α is an action $\alpha'(\vec{p}) : \{e'_1, \dots, e'_n, e_{temp}\}$, where

- e'_i (for $i \in \{1, \dots, n\}$) is obtained from e_i such that $e'_i = Q_{cx}^i \rightsquigarrow \mathbf{add} F_i^{+'} \cup F_i^{-'}$ where:
 - Q_{cx}^i is contextually compiled query of $[q_i^+] \wedge Q_i^-$ w.r.t. \mathbb{D} .
 - for each atom $N(t) \in F_i^+$ (resp. $P(t_1, t_2) \in F_i^+$), we have $N^a(t) \in F_i^{+'}$ (resp. $P^a(t_1, t_2) \in F_i^{+'}$).
 - for each atom $N(t) \in F_i^-$ (resp. $P(t_1, t_2) \in F_i^-$), we have $N^d(t) \in F_i^{-'}$ (resp. $P^d(t_1, t_2) \in F_i^{-'}$).
- $e_{temp} = \{\text{true} \rightsquigarrow \mathbf{add} \{\text{State}(temp)\}\}$

■

Utilizing the notion of delayed action above as well as the other notion introduced earlier, in the following we present how we translate a context-sensitive action invocation into the usual action invocation in S-GKABs. Essentially we express a query as a contextually compiled query, we transform a context expression into the corresponding query, and we translate an action into delayed action.

Definition 6.37 (Action Invocation Obtained From Context-Sensitive Atomic Action Invocation). An *action invocation obtained from context-sensitive atomic action invocation* **pick** $\langle Q(\vec{p}), \varphi_C \rangle . \alpha(\vec{p})$ is an action invocation **pick** $Q'(\vec{p}) . \alpha'(\vec{p})$, where

- $Q' = Q_{cx} \wedge q_{\varphi_C}$ where Q_{cx} is contextually compiled query of Q (see Definition 6.34), and q_{φ_C} is the query that represents the context expression φ_C (see Definition 6.32).
- α' is a delayed action obtained from α (see Definition 6.36).

■

To mimic the context-evolution rules within S-GKABs, we translate them into action invocations as follows:

Definition 6.38 (Action and Action Invocation Obtained From Context-evolution Rule). Let $\mathcal{G}_{cx} = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$ be a CSGKAB. An *action invocation obtained from a context-evolution rule* $\langle Q, \varphi_C \rangle \mapsto C_{new}$ in Π_C , is an action invocation **pick** $Q' . \alpha_C()$ where

1. $Q' = Q_{cx} \wedge q_{\varphi_C}$ where Q_{cx} is contextually compiled query of Q , and q_{φ_C} is the query obtained from the context expression φ_C .
2. α_C is a 0-ary action obtained from $\langle Q, \varphi_C \rangle \mapsto C_{new}$ as follows:
 - (a) For each $[d_i \mapsto v_j] \in C_{new}$, we have:
 - (i) $\text{true} \rightsquigarrow \mathbf{add} \{D_i^{v_j}(\mathbf{c}), \mathbf{D}_i^{v_j}(\mathbf{c})\}$ in $\text{EFF}(\alpha_C)$, and
 - (ii) $\text{true} \rightsquigarrow \mathbf{del} \{D_i^{v_k}(\mathbf{c}), \mathbf{D}_i^{v_k}(\mathbf{c})\}$ in $\text{EFF}(\alpha_C)$ for every $v_k \in \text{Dom}(d_i)$ such that $v_k \neq v_j$.

The intuition is that the effect constructed in the step (i) assigns a new value v_j into d_i while the effects constructed in the step (ii) delete the old value of d_i .

- (b) For each concept name $N \in \text{VOC}(T_{cx})$, we have
 - (i) $N^a(x) \rightsquigarrow \mathbf{add} \{N(x)\}, \mathbf{del} \{N^a(x)\}$ in $\text{EFF}(\alpha_C)$, and
 - (ii) $N^d(x) \rightsquigarrow \mathbf{del} \{N(x), N^d(x)\}$ in $\text{EFF}(\alpha_C)$.

Intuitively, the effects constructed in this step concretely add/delete the ABox assertions that was marked to be added/deleted, and additionally delete all of those markers.

- (c) Similarly for the role names, we create the same effect as in the step (b) above.

In this case we say that α_C is an *action obtained from the context-evolution rule* $\langle Q, \varphi_C \rangle \mapsto C_{new}$.

■

Given a set of context-evolution rules Π_C , we write Λ_C to denote the set of action invocations obtained from all context-evolution rules in Π_C .

In order to simulate the non-deterministic choice of context-evolution rule that change context inside the S-GKABs, we introduce the notion of context-change program as follows.

Definition 6.39 (Context-Change Program). Let Π_C be a set of context-evolution rules and Λ_C be the set of action invocations obtained from Π_C , we define *context-change program* as follows:

Context-Change
Program

$$\delta_{\Pi_C} = \alpha_1 | \dots | \alpha_{|\Lambda_C|}$$

where $\alpha_i \in \Lambda_C$. ■

To check the consistency of the resulting state after the action execution and the context change, we introduce the notion of context-sensitive consistency check action below.

Definition 6.40 (Context-Sensitive Consistency Check Action). Let $\langle T_{cx}, A \rangle$ be a contextualized KB, we define a *context-sensitive consistency check action* $\alpha_{\perp}^{T_{cx}}$ over T_{cx} as a 0-ary (i.e., has no action parameters), where $\text{EFF}(\alpha_{\perp}^{T_{cx}})$ is the smallest set of effects containing:

Context-Sensitive
Consistency Check
Action

- for each functionality assertion $\langle (\text{funct } R) : \varphi_C \rangle \in T_{cx}$, we have
 $q_{\varphi_C} \wedge \exists x, y, z. q_{\text{unsat}}^f((\text{funct } R), x, y, z) \rightsquigarrow \mathbf{add} \{Inc(\mathbf{c})\}$ in $\text{EFF}(\alpha_{\perp}^{T_{cx}})$,
- for each negative concept inclusion assertion $\langle B_1 \sqsubseteq \neg B_2 : \varphi_C \rangle \in T_{cx}$, and for each $C \in \text{CTX}(\mathbb{D})$ we have
 $q_{\varphi_C} \wedge q_C \wedge \text{rew}(\exists x. q_{\text{unsat}}^n(B_1 \sqsubseteq \neg B_2, x), T_{cx}^C) \rightsquigarrow$
 $\mathbf{add} \{Inc(\mathbf{c})\}$ in $\text{EFF}(\alpha_{\perp}^{T_{cx}})$,
- for each negative role inclusion assertion $\langle R_1 \sqsubseteq \neg R_2 : \varphi_C \rangle \in T_{cx}$, and for each $C \in \text{CTX}(\mathbb{D})$ we have
 $q_{\varphi_C} \wedge q_C \wedge \text{rew}(\exists x, y. q_{\text{unsat}}^n(R_1 \sqsubseteq \neg R_2, x, y), T_{cx}^C) \rightsquigarrow$
 $\mathbf{add} \{Inc(\mathbf{c})\}$ in $\text{EFF}(\alpha_{\perp}^{T_{cx}})$,
- additionally, we have $\mathbf{true} \rightsquigarrow \mathbf{del} \text{State}(temp)$ in $\text{EFF}(\alpha_{\perp}^{T_{cx}})$

where q_{φ_C} is a query that represents the context expression φ_C , q_C is a query that represents context C , and we also make use the abbreviations of FOL query in Definition 2.42. ■

For brevity, in this section we simply say consistency check action instead of context-sensitive consistency check action.

We now define a translation function κ_{cx} that essentially concatenates each action invocation with a program that non-deterministically choose an action that changes the context, and also an action that checks the inconsistency. Additionally, the translation function κ_{cx} also serves as a one-to-one correspondence (bijection) between the original and the translated program (as well as between the sub-program).

Definition 6.41 (Program Translation κ_{cx}). Given a S-CSGKAB $\mathcal{G}_{cx} = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$, we define a *translation* κ_{cx} which translates the program δ inductively as follows:

Program Translation
 κ_{cx}

- $\kappa_{cx}(\mathbf{pick} \langle Q(\vec{p}), \varphi_C \rangle. \alpha(\vec{p})) = \mathbf{pick} Q'(\vec{p}). \alpha'(\vec{p}) ; \delta_{\Pi_C} ; \mathbf{pick} \mathbf{true}. \alpha_{\perp}^{T_{cx}}()$
- $\kappa_{cx}(\varepsilon) = \varepsilon$
- $\kappa_{cx}(\delta_1 | \delta_2) = \kappa_{cx}(\delta_1) | \kappa_{cx}(\delta_2)$
- $\kappa_{cx}(\delta_1 ; \delta_2) = \kappa_{cx}(\delta_1) ; \kappa_{cx}(\delta_2)$

- $\kappa_{cx}(\text{if } \varphi \text{ then } \delta_1 \text{ else } \delta_2) = \text{if } \varphi \text{ then } \kappa_{cx}(\delta_1) \text{ else } \kappa_{cx}(\delta_2)$
- $\kappa_{cx}(\text{while } \varphi \text{ do } \delta) = \text{while } \varphi \text{ do } \kappa_{cx}(\delta)$

where

- **pick** $Q'(\vec{p}).\alpha'(\vec{p})$ is an action invocation obtained from **pick** $\langle Q(\vec{p}), \varphi_C \rangle.\alpha(\vec{p})$ (see Definition 6.37),
- δ_{Π_C} is a context-change program obtained from Π_C (see Definition 6.39),
- $\alpha_{\perp}^{T_{cx}}$ is a consistency check action (see Definition 6.40).

■

To transform S-CSGKABs into the corresponding S-GKABs, we define a translation τ_{cx} that, given an S-CSGKAB, generates an S-GKAB as follows.

Translation from
S-CSGKAB to
S-GKAB

Definition 6.42 (Translation from S-CSGKAB to S-GKAB). We define a translation τ_{cx} that, given an S-CSGKAB $\mathcal{G}_{cx} = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$, produces an S-GKAB $\tau_{cx}(\mathcal{G}_{cx}) = \langle T', A'_0, \Gamma', \delta' \rangle$, where

- $T' = \{Inc \sqsubseteq \neg Inc\} \cup T_{\mathbb{D}}$, where $T_{\mathbb{D}}$ is a TBox obtained from a set of context dimensions \mathbb{D} ,
- $A'_0 = A_0 \cup A_{C_0}$ (where A_{C_0} is an ABox obtained from C_0),
- $\Gamma' = \Gamma_{\alpha} \cup \Gamma_C$ where:
 - Γ_{α} is obtained from Γ such that for each action $\alpha \in \Gamma$, we have $\alpha' \in \Gamma_{\alpha}$ where α' is a delayed action obtained from α (see Definition 6.36).
 - Γ_C is obtained from Π_C such that for each context-evolution rule $\langle Q, \varphi_C \rangle \mapsto C_{new}$ in Π_C , we have $\alpha_C \in \Gamma_C$ where α_C is an action obtained from the context-evolution rule $\langle Q, \varphi_C \rangle \mapsto C_{new}$ (see Definition 6.38).
- $\delta' = \kappa_{cx}(\delta)$.

■

A $\mu\mathcal{L}_{CTX}$ property Φ over CSGKABs \mathcal{G}_{cx} can then be recast as a corresponding $\mu\mathcal{L}_A^{EQL}$ property over S-GKAB $\tau_{cx}(\mathcal{G}_{cx})$ by simply substituting each subformula $\langle \rightarrow \rangle \Psi$ of Φ with $\langle \rightarrow \rangle \langle \rightarrow \rangle \langle \rightarrow \rangle \Psi$ (similarly for $[-] \Phi$). Formally we define such formula translation as follows:

$\mu\mathcal{L}_{CTX}$ Formula
Translation t_{trip}

Definition 6.43 (Translation t_{trip}). We define a translation t_{trip} that takes a $\mu\mathcal{L}_{CTX}$ formula Φ as an input and produces a new $\mu\mathcal{L}_{CTX}$ formula $t_{trip}(\Phi)$ by recurring over the structure of Φ as follows:

- $t_{trip}(Q) = Q_{cx}$
- $t_{trip}(\varphi_C) = q_{\varphi_C}$
- $t_{trip}(\neg \Phi) = \neg t_{trip}(\Phi)$
- $t_{trip}(\exists x. \Phi) = \exists x. t_{trip}(\Phi)$
- $t_{trip}(\Phi_1 \vee \Phi_2) = t_{trip}(\Phi_1) \vee t_{trip}(\Phi_2)$
- $t_{trip}(\mu Z. \Phi) = \mu Z. t_{trip}(\Phi)$
- $t_{trip}(\langle \rightarrow \rangle \Phi) = \langle \rightarrow \rangle \langle \rightarrow \rangle \langle \rightarrow \rangle t_{trip}(\Phi)$

where Q_{cx} is a contextually compiled query of Q (see Definition 6.34), and q_{φ_C} is the query that represents the context expression φ_C (see Definition 6.32).

■

With this translation in hand, we will show later that $\mathcal{Y}_{\mathcal{G}_{cx}}^{f_{cx}^S} \models \Phi$ if and only if $\mathcal{Y}_{\tau_{cx}(\mathcal{G}_{cx})}^{f_{cx}^S} \models t_{trip}(\Phi)$, which consequently means that the verification of $\mu\mathcal{L}_{CTX}$ over S-CSGKABs can be reduced to the corresponding verification over S-GKABs. The core idea of the proof is to use a certain bisimulation relation in which two bisimilar transition systems (w.r.t. this bisimulation relation) can not be distinguished by $\mu\mathcal{L}_{CTX}$ properties modulo the formula translation t_{trip} . Then, we show that the transition system of an S-CSGKAB is bisimilar to the transition system of its corresponding S-GKAB w.r.t. this bisimulation relation.

6.5.2.2 Skip-two Bisimulation (ST-Bisimulation)

Towards defining the notion of ST-Bisimulation, we introduce the notion of contextually equal between a state of context-sensitive transition system and a state of KB transition system as follows:

Definition 6.44 (Contextually Equal State).

Contextually Equal State

Let $\mathcal{T}_1 = \langle \Delta, T_{cx}, \Sigma_1, s_{01}, abox_1, ctx, \Rightarrow_1 \rangle$ be context-sensitive transition system, and $\mathcal{T}_2 = \langle \Delta, T, \Sigma_2, s_{02}, abox_2, \Rightarrow_2 \rangle$ be KB transition systems. Consider two states $s_1 \in \Sigma_1$ and $s_2 \in \Sigma_2$, we say s_1 is *contextually equal* to s_2 , written $s_1 =_{cx} s_2$ if $abox_1(s_1) \cup A_{ctx(s_1)} = abox_2(s_2)$. ■

Intuitively, two contextually equal states contain the same data/facts in the ABox and also have the same context information (although they are encoded in a different way). We then define the notion of ST-Bisimulation as follows:

Definition 6.45 (Skip-two Bisimulation (ST-Bisimulation)).

Skip-two Bisimulation (ST-Bisimulation)

Let $\mathcal{T}_1 = \langle \Delta, T_{cx}, \Sigma_1, s_{01}, abox_1, ctx, \Rightarrow_1 \rangle$ be a context-sensitive transition system, and $\mathcal{T}_2 = \langle \Delta, T, \Sigma_2, s_{02}, abox_2, \Rightarrow_2 \rangle$ be a KB transition system, with $\text{ADOM}(abox_1(s_{01})) \subseteq \Delta$ and $\text{ADOM}(abox_2(s_{02})) \subseteq \Delta$. A *skip-two bisimulation* (ST-Bisimulation) between \mathcal{T}_1 and \mathcal{T}_2 is a relation $\mathcal{B} \subseteq \Sigma_1 \times \Sigma_2$ such that $\langle s_1, s_2 \rangle \in \mathcal{B}$ implies that:

1. $s_1 =_{cx} s_2$
2. for each s'_1 , if $s_1 \Rightarrow_1 s'_1$ then there exists t_1, t_2 , and s'_2 with

$$s_2 \Rightarrow_2 t_1 \Rightarrow_2 t_2 \Rightarrow_2 s'_2$$

such that $\langle s'_1, s'_2 \rangle \in \mathcal{B}$, $\text{State}(temp) \notin abox_2(s'_2)$ and $\text{State}(temp) \in abox_2(t_i)$ for $i \in \{1, 2\}$.

3. for each s'_2 , if

$$s_2 \Rightarrow_2 t_1 \Rightarrow_2 t_2 \Rightarrow_2 s'_2$$

with $\text{State}(temp) \in abox_2(t_i)$ for $i \in \{1, 2\}$ and $\text{State}(temp) \notin abox_2(s'_2)$, then there exists s'_1 with $s_1 \Rightarrow_1 s'_1$, such that $\langle s'_1, s'_2 \rangle \in \mathcal{B}$. ■

Let $\mathcal{T}_1 = \langle \Delta, T_{cx}, \Sigma_1, s_{01}, abox_1, ctx, \Rightarrow_1 \rangle$ be a context-sensitive transition system, and $\mathcal{T}_2 = \langle \Delta, T, \Sigma_2, s_{02}, abox_2, \Rightarrow_2 \rangle$ be a KB transition system, a state $s_1 \in \Sigma_1$ is *ST-bisimilar* to $s_2 \in \Sigma_2$, written $s_1 \sim_{ST} s_2$, if there exists an ST-bisimulation relation \mathcal{B} between \mathcal{T}_1 and \mathcal{T}_2 such that $\langle s_1, s_2 \rangle \in \mathcal{B}$. A transition system \mathcal{T}_1 is *ST-bisimilar* to \mathcal{T}_2 , written $\mathcal{T}_1 \sim_{ST} \mathcal{T}_2$, if there exists an ST-bisimulation relation \mathcal{B} between \mathcal{T}_1 and \mathcal{T}_2 such that $\langle s_{01}, s_{02} \rangle \in \mathcal{B}$.

In the following two lemmas we show some important properties of ST-bisimilar states and transition systems that will be useful later to show that we can recast the verification of S-CSGKABs into S-GKABs.

Lemma 6.46. *Let $\Upsilon_1 = \langle \Delta, T_{cx}, \Sigma_1, s_{01}, abox_1, ctx, \Rightarrow_1 \rangle$ be a context-sensitive transition system, and $\Upsilon_2 = \langle \Delta, T, \Sigma_2, s_{02}, abox_2, \Rightarrow_2 \rangle$ be a KB transition system. Consider two states $s_1 \in \Sigma_1$ and $s_2 \in \Sigma_2$ such that $s_1 \sim_{ST} s_2$. Then for every formula Φ of $\mu\mathcal{L}_{CTX}$, and every valuations v_1 and v_2 that assign to each of its free variables a constant $c_1 \in \text{ADOM}(abox_1(s_1))$ and $c_2 \in \text{ADOM}(abox_2(s_2))$, such that $c_1 = c_2$, we have that*

$$\Upsilon_1, s_1 \models \Phi v_1 \text{ if and only if } \Upsilon_2, s_2 \models t_{trip}(\Phi) v_2.$$

Proof. In general, the proof is similar to the proof of Lemma 5.41. Thus, here we only highlight some interesting cases in the induction.

- Case of $\Phi = Q$: Since $s_1 \sim_{ST} s_2$, we have $s_1 =_{cx} s_2$. Hence, by Definition 6.44, we have $abox_1(s_1) \cup A_{ctx(s_1)} = abox_2(s_2)$, and furthermore, by Lemma 6.35, we have

$$\text{CERT}(Q, T_{cx}^{ctx(s_1)}, abox(s_1)) = \text{CERT}(Q_{cx}, T_D, abox(s_2))$$

Since $t_{trip}(Q) = Q_{cx}$, it is easy to see that for every valuations v_1 and v_2 that assign to each of its free variables a constant $c_1 \in \text{ADOM}(abox_1(s_1))$ and $c_2 \in \text{ADOM}(abox_2(s_2))$, such that $c_1 = c_2$, we have

$$\Upsilon_1, s_1 \models Q v_1 \text{ if and only if } \Upsilon_2, s_2 \models t_{trip}(Q) v_2.$$

- Case of $\Phi = \langle \neg \rangle \Psi$: Assume $\Upsilon_1, s_1 \models (\langle \neg \rangle \Psi) v_1$, then there exists s'_1 s.t. $s_1 \Rightarrow_1 s'_1$ and $\Upsilon_1, s'_1 \models \Psi v_1$. Since $s_1 \sim_{ST} s_2$, there exist t_1, t_2 and s'_2 s.t.

$$s_2 \Rightarrow_2 t_1 \Rightarrow_2 t_2 \Rightarrow_2 s'_2$$

and $s'_1 \sim_{ST} s'_2$. Hence, by induction hypothesis, for every valuations v_2 that assign to each free variables x of $t_{trip}(\Psi)$ a constant $c_2 \in \text{ADOM}(abox_2(s_2))$, such that $c_2 = c_1$ with $x/c_1 \in v_1$, we have

$$\Upsilon_2, s'_2 \models t_{trip}(\Psi) v_2.$$

Since $s_2 \Rightarrow_2 t_1 \Rightarrow_2 t_2 \Rightarrow_2 s'_2$, therefore we get

$$\Upsilon_2, s_2 \models (\langle \neg \rangle \langle \neg \rangle \langle \neg \rangle t_{trip}(\Psi)) v_2.$$

Since $t_{trip}(\langle \neg \rangle \Phi) = \langle \neg \rangle \langle \neg \rangle \langle \neg \rangle t_{trip}(\Phi)$, we therefore have

$$\Upsilon_2, s_2 \models t_{trip}(\langle \neg \rangle \Psi) v_2.$$

The other direction can be shown in a similar way. □

Lemma 6.47. *Consider a context-sensitive transition system Υ_1 and a KB transition system Υ_2 such that $\Upsilon_1 \sim_{ST} \Upsilon_2$. For every closed $\mu\mathcal{L}_{CTX}$ formula Φ , we have:*

$$\Upsilon_1 \models \Phi \text{ if and only if } \Upsilon_2 \models t_{trip}(\Phi)$$

Proof. Let $\mathcal{R}_1 = \langle \Delta, T_{cx}, \Sigma_1, s_{01}, abox_1, ctx, \Rightarrow_1 \rangle$, and $\mathcal{R}_2 = \langle \Delta, T, \Sigma_2, s_{02}, abox_2, \Rightarrow_2 \rangle$. By the definition of ST-bisimilar transition system we have that $s_{01} \sim_{ST} s_{02}$. Thus, we obtain the proof as a consequence of Lemma 6.46, due to the fact that

$$\mathcal{R}_1, s_{01} \models \Phi \text{ if and only if } \mathcal{R}_2, s_{02} \models t_{trip}(\Phi)$$

□

6.5.2.3 Reducing the Verification of S-CSGKABs into S-GKABs

We now step forward to show that we can recast the verification of S-CSGKABs into S-GKABs. We open this section by showing that the transition systems of an S-CSGKAB and its corresponding S-GKAB (obtained through τ_{cx}) are ST-bisimilar. The following two lemmas are aimed to show this fact.

Lemma 6.48. *Let \mathcal{G}_{cx} be an S-CSGKAB with transition system $\mathcal{R}_{\mathcal{G}_{cx}}^{f_{cx}^{cs}}$, and let $\tau_{cx}(\mathcal{G}_{cx})$ be its corresponding S-GKAB (with transition system $\mathcal{R}_{\tau_{cx}(\mathcal{G}_{cx})}^{f_s^{cs}}$) obtained through τ_{cx} . Consider a state $s_{cx} = \langle A_{cx}, m_{cx}, C, \delta_{cx} \rangle$ of $\mathcal{R}_{\mathcal{G}_{cx}}^{f_{cx}^{cs}}$ and a state $s_s = \langle A_s, m_s, \delta_s \rangle$ of $\mathcal{R}_{\tau_{cx}(\mathcal{G}_{cx})}^{f_s^{cs}}$. If $s_{cx} =_{cx} s_s$, $m_{cx} = m_s$ and $\delta_s = \kappa_{cx}(\delta_{cx})$, then $\langle A_{cx}, m_{cx}, C, \delta_{cx} \rangle \sim_{ST} \langle A_s, m_s, \delta_s \rangle$.*

Proof. For the simplicity of the proof, here we ignore the presence of the ABox assertion **State**(temp) that acts as a special marker and marks the intermediate states. The important thing to observe is that **State**(temp) is always added to the intermediate state (where we still need to change the context and do the inconsistency check) but then it will be deleted after that. Furthermore, the presence of **State**(temp) also distinguish the intermediate and stable state. Now, let

1. $\mathcal{G}_{cx} = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$, and $\mathcal{R}_{\mathcal{G}_{cx}}^{f_{cx}^{cs}} = \langle \Delta, T_{cx}, \Sigma_{cx}, s_{0cx}, abox_{cx}, ctx, \Rightarrow_{cx} \rangle$,
2. $\tau_{cx}(\mathcal{G}_{cx}) = \langle T', A'_0, \Gamma', \delta' \rangle$ and $\mathcal{R}_{\tau_{cx}(\mathcal{G}_{cx})}^{f_s^{cs}} = \langle \Delta, T', \Sigma_s, s_{0s}, abox_s, \Rightarrow_s \rangle$.

Now, we have to show the following: For every state $\langle A'''_{cx}, m'''_{cx}, C''', \delta'''_{cx} \rangle$ such that

$$\langle A_{cx}, m_{cx}, C, \delta_{cx} \rangle \Rightarrow \langle A'''_{cx}, m'''_{cx}, C''', \delta'''_{cx} \rangle,$$

there exists states $\langle A'_s, m'_s, \delta'_s \rangle$, $\langle A''_s, m''_s, \delta''_s \rangle$, and $\langle A'''_s, m'''_s, \delta'''_s \rangle$ such that:

- (a) we have $\langle A_s, m_s, \delta_s \rangle \Rightarrow_s \langle A'_s, m'_s, \delta'_s \rangle \Rightarrow_s \langle A''_s, m''_s, \delta''_s \rangle \Rightarrow_s \langle A'''_s, m'''_s, \delta'''_s \rangle$
- (b) $\langle A'''_{cx}, m'''_{cx}, C''', \delta'''_{cx} \rangle =_{cx} \langle A'''_s, m'''_s, \delta'''_s \rangle$
- (c) $m'''_s = m'''_{cx}$;
- (d) $\delta'''_s = \kappa_C(\delta'''_{cx})$.

By definition of $\mathcal{R}_{\mathcal{G}_{cx}}^{f_{cx}^{cs}}$, since $\langle A_{cx}, m_{cx}, C, \delta_{cx} \rangle \Rightarrow \langle A'''_{cx}, m'''_{cx}, C''', \delta'''_{cx} \rangle$, we have $\langle A_{cx}, m_{cx}, C, \delta_{cx} \rangle \xrightarrow{\alpha\sigma_{cx}, f_C} \langle A'''_{cx}, m'''_{cx}, C''', \delta'''_{cx} \rangle$. Hence, by the definition of $\xrightarrow{\alpha\sigma_{cx}, f_C}$, we have that:

- $\langle \langle A_{cx}, m_{cx}, C \rangle, \alpha\sigma_{cx}, \langle A'''_{cx}, m'''_{cx}, C''' \rangle \rangle \in \text{CS-TELL}_{f_{cx}^{cs}}$, and
- σ_{cx} is a legal parameter assignment for α in A_{cx} w.r.t. context C and action invocation **pick** $\langle Q(\vec{p}), \varphi_C \rangle. \alpha(\vec{p})$ (i.e., $\text{ASK}(Q\sigma_{cx}, T_{cx}^C, A_{cx}) = \text{true}$). Notice that w.l.o.g. **pick** $\langle Q(\vec{p}), \varphi_C \rangle. \alpha(\vec{p})$ is the next instruction that should be executed in δ_{cx} .
- $C \cup \Phi_D \models \varphi_C$.

Since $\langle \langle A_{cx}, m_{cx}, C \rangle, \alpha\sigma_{cx}, \langle A_{cx}''', m_{cx}''', C''' \rangle \rangle \in \text{CS-TELL}_{f_S^{cx}}$, by the definition of $\text{CS-TELL}_{f_S^{cx}}$, we have:

1. $\langle A_{cx}, C, C''' \rangle \in \text{CTX-CHG}$,
2. there exists $\theta_{cx} \in \text{EVAL}(\text{ADD}(T_{cx}^C, A_{cx}, \alpha\sigma_{cx}))$ such that:
 - a) θ_{cx} and m_{cx} agree on the common values in their domains.
 - b) $m_{cx}''' = m_{cx} \cup \theta_{cx}$;
 - c) $\langle A_{cx}, \text{ADD}(T_{cx}^C, A_{cx}, \alpha\sigma_{cx})\theta_{cx}, \text{DEL}(T_{cx}^C, A_{cx}, \alpha\sigma_{cx}), C''', A_{cx}''' \rangle \in f_S^{cx}$;
 - d) A_{cx} is T_{cx}^C -consistent, and A_{cx}''' is $T_{cx}^{C'''}-consistent$.

Since $\langle A_{cx}, \text{ADD}(T_{cx}^C, A_{cx}, \alpha\sigma_{cx})\theta_{cx}, \text{DEL}(T_{cx}^C, A_{cx}, \alpha\sigma_{cx}), C''', A_{cx}''' \rangle \in f_S^{cx}$, by the definition of f_S^{cx} , we have $A_{cx}''' = (A_{cx} \setminus \text{DEL}(T_{cx}^C, A_{cx}, \alpha\sigma_{cx})) \cup \text{ADD}(T_{cx}^C, A_{cx}, \alpha\sigma_{cx})\theta_{cx}$. Furthermore, since $\delta_s = \kappa_{cx}(\delta_{cx})$, by the definition of κ_{cx} , we have that

$$\kappa_{cx}(\text{pick } \langle Q(\vec{p}), \varphi_C \rangle . \alpha(\vec{p})) = \text{pick } Q'(\vec{p}) . \alpha'(\vec{p}) ; \delta_{\Pi_C} ; \text{pick true} . \alpha_{\perp}^{T_{cx}}()$$

Hence, the next executable part of the program on state $\langle A_s, m_s, \delta_s \rangle$ is

$$\text{pick } Q'(\vec{p}) . \alpha'(\vec{p}) ; \delta_{\Pi_C} ; \text{pick true} . \alpha_{\perp}^{T_{cx}}().$$

Notice that

- $\text{pick } Q'(\vec{p}) . \alpha'(\vec{p})$ is obtained from $\text{pick } \langle Q(\vec{p}), \varphi_C \rangle . \alpha(\vec{p})$, and
- since $s_{cx} =_{cx} s_s$, then we have that $A_s = A_{cx} \cup A_C$, where A_C is the set of ABox assertion that represents the context C (see Definition 6.31).

Thus, by Definition 6.37, we have that $Q' = Q_{cx} \wedge q_{\varphi_C}$. Since $C \cup \Phi_{\mathbb{D}} \models \varphi_C$ and q_{φ_C} only use context dimension concept, by Lemma 6.33, it is easy to see that $\text{CERT}(q_{\varphi_C}, T', A_s) = \text{true}$. Furthermore, by Lemma 6.35, we have that $\text{CERT}(Q, T_{cx}^C, A_{cx}) = \text{CERT}(Q_{cx}, T', A_s)$. Therefore, now we can construct σ_s that maps parameters of α' to constants in $\text{ADOM}(A_s)$ such that $\sigma_c = \sigma_s$.

Now, since we have $m_s = m_{cx}$, then we can construct θ_s such that $\theta_s = \theta_{cx}$. Hence, we have the following:

- θ_s and m_s agree on the common values in their domains.
- $m_s''' = \theta_s \cup m_s = \theta_{cx} \cup m_{cx} = m_{cx}'''$.

Now, let $A'_s = A_s \cup \text{ADD}(T', A_s, \alpha'\sigma_s)\theta_s$, i.e., A'_s captures the result of the execution of action α' and by the definition of delayed action α' (see Definition 6.36). Considering the form of T' , it is easy to see that A'_s is T' -consistent. Thus, by the definition of TELL_{f_s} , we have $\langle \langle A_s, m_s \rangle, \alpha'\sigma_s, \langle A'_s, m_s''' \rangle \rangle \in \text{TELL}_{f_s}$. Moreover, we have

$$\langle A_s, m_s, \text{pick } Q(\vec{p}) . \alpha'(\vec{p}) ; \delta_0 \rangle \xrightarrow{\alpha'\sigma_s, f_s} \langle A'_s, m_s''', \delta_0 \rangle$$

where $\delta_0 = \delta_{\Pi_C} ; \text{pick true} . \alpha_{\perp}^{T_{cx}}()$, and $A_{cx} \cup A_C \subseteq A'_s$ (notice that, by construction, α' only adds new ABox assertions). W.l.o.g., let $A'_s = A_{cx} \cup A_{\alpha'} \cup A_C$ (i.e., $A_{\alpha'}$ represents the set of ABox assertions that was just added by α').

Now, since $\langle A_{cx}, C, C''' \rangle \in \text{CTX-CHG}$, by Definition 6.16, there exists a context-evolution rule $\langle Q, \varphi_C \rangle \mapsto C_{new}$ in Π_C s.t.:

1. $\text{ASK}(Q, T_{cx}^C, A_{cx})$ is true;
2. $C \cup \Phi_{\mathbb{D}} \models \varphi_C$;
3. for every context dimension $d \in \mathbb{D}$ s.t. $[d \rightsquigarrow v] \in C_{new}$, we have $[d \rightsquigarrow v] \in C'''$;

4. for every context dimension $d \in \mathbb{D}$ s.t. $[d \rightsquigarrow v] \in C$, and there does not exist any v_2 s.t. $[d \rightsquigarrow v_2] \in C_{new}$, we have $[d \rightsquigarrow v] \in C'''$.

Additionally, by the definition of δ_{Π_C} (see Definition 6.39), it is easy to see that there exists A'_s such that we have

$$\langle A'_s, m_s''', \delta_0 \rangle \xrightarrow{\alpha_C \sigma'_s, f_s} \langle A''_s, m_s''', \delta_1 \rangle$$

where σ'_s is an empty substitution, $A''_s = A'''_{cx} \cup A_{C'''}_s$, and $\delta_1 = \mathbf{pick\ true}.\alpha_{\perp}^{T_{cx}}()$.

Now, notice that $\alpha_{\perp}^{T_{cx}}$ only change the ABox when there is an inconsistency, since A'''_{cx} is $T_{cx}^{C'''}_s$ -consistent, it is easy to see that we have

$$\langle A''_s, m_s''', \delta_1 \rangle \xrightarrow{\alpha_{\perp}^{T_{cx}} \sigma''_s, f_s} \langle A'''_s, m_s''', \delta'''_s \rangle$$

where $A'''_s = A''_s$, and $\langle A'''_{cx}, m'''_{cx}, C''', \delta'''_{cx} \rangle =_{cx} \langle A'''_s, m_s''', \delta'''_s \rangle$.

The other direction of bisimulation relation can be proven in a similar way. \square

Having Lemma 6.48 in hand, we can easily show that given an S-CSGKAB, its transition system is ST-bisimilar to the transition of its corresponding S-GKAB that is obtained via the translation τ_{cx} as follows.

Lemma 6.49. *Given an S-CSGKAB \mathcal{G}_{cx} with transition system $\Upsilon_{\mathcal{G}_{cx}}^{f_{cx}^{cx}}$, let $\tau_{cx}(\mathcal{G}_{cx})$ be the corresponding S-GKAB (with transition system $\Upsilon_{\tau_{cx}(\mathcal{G}_{cx})}^{f_s}$) obtained from \mathcal{G}_{cx} via τ_{cx} . We have $\Upsilon_{\mathcal{G}_{cx}}^{f_{cx}^{cx}} \sim_{ST} \Upsilon_{\tau_{cx}(\mathcal{G}_{cx})}^{f_s}$.*

Proof. Let

1. $\mathcal{G}_{cx} = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$, and

$$\Upsilon_{\mathcal{G}_{cx}}^{f_{cx}^{cx}} = \langle \Delta, T_{cx}, \Sigma_{cx}, s_{0_{cx}}, abox_{cx}, ctx, \Rightarrow_{cx} \rangle,$$

2. $\tau_{cx}(\mathcal{G}_{cx}) = \langle T', A'_0, \Gamma', \delta' \rangle$ and $\Upsilon_{\tau_{cx}(\mathcal{G}_{cx})}^{f_s} = \langle \Delta, T_s, \Sigma_s, s_{0_s}, abox_s, \Rightarrow_s \rangle$.

We have that $s_{0_{cx}} = \langle A_0, m_{cx}, C_0, \delta \rangle$ and $s_{0_s} = \langle A'_0, m_s, \delta' \rangle$ where $m_{cx} = m_s = \emptyset$. By the definition of κ_{cx} and τ_{cx} , we also have $s_{0_{cx}} =_{cx} s_{0_s}$, and $\delta' = \kappa_{cx}(\delta)$. Hence, by Lemma 6.48, we have $s_{0_{cx}} \sim_{ST} s_{0_s}$. Therefore, by the definition of ST-bisimulation, we have $\Upsilon_{\mathcal{G}_{cx}}^{f_{cx}^{cx}} \sim_{ST} \Upsilon_{\tau_{cx}(\mathcal{G}_{cx})}^{f_s}$. \square

Having all of these machinery in hand, we are now ready to show that the verification of $\mu\mathcal{L}_{CTX}$ properties over S-CSGKABs can be recast as verification of $\mu\mathcal{L}_A^{EQL}$ over S-GKABs as follows.

Theorem 6.50. *Given an S-CSGKAB \mathcal{G}_{cx} and a closed $\mu\mathcal{L}_{CTX}$ property Φ , we have*

$$\Upsilon_{\mathcal{G}_{cx}}^{f_{cx}^{cx}} \models \Phi \text{ if and only if } \Upsilon_{\tau_{cx}(\mathcal{G}_{cx})}^{f_s} \models t_{trip}(\Phi)$$

Proof. By Lemma 6.49, we have that $\Upsilon_{\mathcal{G}_{cx}}^{f_{cx}^{cx}} \sim_{ST} \Upsilon_{\tau_{cx}(\mathcal{G}_{cx})}^{f_s}$. Hence, by Lemma 6.47, we have that for every $\mu\mathcal{L}_{CTX}$ property Φ

$$\Upsilon_{\mathcal{G}_{cx}}^{f_{cx}^{cx}} \models \Phi \text{ if and only if } \Upsilon_{\tau_{cx}(\mathcal{G}_{cx})}^{f_s} \models t_{trip}(\Phi)$$

\square

6.5.3 Verification of Run-Bounded Standard CSGKABs

An interesting property of the translation τ_{cx} is that it preserves run-boundedness.

Lemma 6.51. *Let \mathcal{G}_{cx} be an S-CSGKAB and $\tau_{cx}(\mathcal{G}_{cx})$ be its corresponding S-GKAB. We have \mathcal{G}_{cx} is run-bounded if and only if $\tau_{cx}(\mathcal{G}_{cx})$ is run-bounded.*

Proof. Let

1. $\mathcal{R}_{\mathcal{G}_{cx}}^{fs}$ be the transition system of \mathcal{G}_{cx} .
2. $\mathcal{R}_{\tau_{cx}(\mathcal{G}_{cx})}^{fs}$ be the transition system of $\tau_{cx}(\mathcal{G}_{cx})$.

The proof is easily obtained due to the following facts:

- the program that is used to simulate the context evolution does not inject unbounded number of new constants. In fact, we only reserve a constant \mathbf{c} to simulate the context (i.e., to construct the ABox assertions that represent the context dimension assignments).
- similarly, the action that is used to check the inconsistency does not introduce unbounded number of new constants.
- by Lemma 6.49, we have that $\mathcal{R}_{\mathcal{G}_{cx}}^{fs} \sim_{ST} \mathcal{R}_{\tau_{cx}(\mathcal{G}_{cx})}^{fs}$. Thus, basically they are “equivalent” modulo intermediate states (states containing $\text{State}(\text{temp})$), and each two bisimilar states are equivalent modulo context ABox assertions.

□

Now, we can easily acquire the following result on verification of $\mu\mathcal{L}_{\text{CTX}}$ properties over run-bounded S-CSGKABs.

Theorem 6.52 (Verification of Run-Bounded S-CSGKABs). *Verification of closed $\mu\mathcal{L}_{\text{CTX}}$ formulas over a run-bounded S-CSGKAB is decidable and can be reduced to finite-state model checking.*

Proof. From Theorem 6.50, Lemma 6.51, and Theorem 4.54 we have that verification of closed $\mu\mathcal{L}_{\text{CTX}}$ formulas over run-bounded S-CSGKABs can be reduced to the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ formulas over run-bounded KABs. Then, by Theorem 3.30, we have that verification of $\mu\mathcal{L}_A^{\text{EQL}}$ over run-bounded DCDS is decidable and can be reduced to finite-state model checking. □

6.6 Capturing Standard GKABs within Standard CSGKABs

So far we have seen that we can compile S-CSGKABs into S-GKABs and recast the verification of S-CSGKABs into S-GKABs. Now, we show that we can actually capture S-GKABs within S-CSGKABs. As a consequence, we have that S-GKABs and S-CSGKABs are essentially reducible to each other in terms of verification.

The idea to capture S-GKABs within S-CSGKABs is as follows:

- We introduce only a single context dimension and it has only a single possible value. Thus we basically can only have one possible context.
- We transform the TBox in the given S-GKAB into a contextualized TBox such that each assertion holds in our only one possible context.
- We introduce a single context-evolution rule that never change the context (keep the context stay the same).

- The initial context will be our only one possible context.
- Each action invocation in the program of the given S-GKAB is translated into a context-sensitive action invocation where the corresponding context expression is our only one possible context dimension assignment. Thus this context expression will not affect the action execution since it will always holds.

To formalize the ideas above, in the following we fix a set \mathbb{D} of context dimension containing only a single context dimension d (i.e., $\mathbb{D} = \{d\}$). Moreover, $d \in \mathbb{D}$ has a tree shaped finite value domain $\langle \text{Dom}(d), \prec_d \rangle$ where $\text{Dom}(d)$ contains only a single value \top_d (i.e., $\text{Dom}(d) = \top_d$).

We now introduce the translation for program in S-GKABs. In particular, we define a translation function κ_{sc} that basically replaces each action invocation with a context-sensitive action invocation in which its context expression always holds in any context. Additionally, the translation function κ_{sb} also serves as a one-to-one correspondence (bijection) between the original and the translated program (as well as between the sub-program).

Definition 6.53 (Program Translation κ_{sc}). Given a set of actions Γ , a program δ over Γ , and a TBox T , we define a *translation* κ_{sc} which translates a program into a program inductively as follows:

Program Translation
 κ_{sc}

$$\begin{aligned}
\kappa_{sc}(\text{pick } Q(\vec{p}).\alpha(\vec{p})) &= \text{pick } \langle Q(\vec{p}), \varphi_C \rangle.\alpha(\vec{p}) \\
\kappa_{sc}(\varepsilon) &= \varepsilon \\
\kappa_{sc}(\delta_1 | \delta_2) &= \kappa_{sc}(\delta_1) | \kappa_{sc}(\delta_2) \\
\kappa_{sc}(\delta_1 ; \delta_2) &= \kappa_{sc}(\delta_1) ; \kappa_{sc}(\delta_2) \\
\kappa_{sc}(\text{if } \varphi \text{ then } \delta_1 \text{ else } \delta_2) &= \text{if } \varphi \text{ then } \kappa_{sc}(\delta_1) \text{ else } \kappa_{sc}(\delta_2) \\
\kappa_{sc}(\text{while } \varphi \text{ do } \delta) &= \text{while } \varphi \text{ do } \kappa_{sc}(\delta)
\end{aligned}$$

where $\varphi_C = \{[d \rightsquigarrow \top_d]\}$ ■

Having the necessary ingredients, we define the following translation that transform S-GKABs into S-CSGKABs as follows.

Definition 6.54 (Translation from S-GKAB to S-CSGKAB). We define a translation τ_{sc} that, given an S-GKAB $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$, produces an S-CSGKAB $\tau_{sc}(\mathcal{G}) = \langle T_{cx}, A_0, \Gamma, \delta', C_0, \Pi_C \rangle$, where

Translation from
S-GKAB to
S-CSGKAB

- T_{cx} is obtained from T such that for each TBox assertion $t \in T$ we have $\langle t : \varphi \rangle$ where $\varphi = [d \rightsquigarrow \top_d]$,
- $\delta' = \kappa_{sc}(\delta)$.
- $C_0 = \{[d \rightsquigarrow \top_d]\}$,
- $\Pi_C = \{\langle \text{true}, [d \rightsquigarrow \top_d] \rangle \mapsto \{[d \rightsquigarrow \top_d]\}\}$ ■

We now proceed to show that given an S-GKAB \mathcal{G} , and $\mu\mathcal{L}_A^{\text{EQL}}$ formula Φ , we have that $\mathcal{R}_{\mathcal{G}}^{fs} \models \Phi$ if and only if $\mathcal{R}_{\tau_{sc}(\mathcal{G})}^{fs} \models \Phi$. The strategy is as follows:

1. Recall the notion of E-Bisimulation in Section 4.3.1. Here we use a similar notion of bisimulation except that now the bisimulation relation is defined between a KB transition system and a context-sensitive transition system. However, the

- bisimulation condition are kept the same. Therefore, for brevity, here we do not redefine a new bisimulation relation. All notions related to E-Bisimulation that was introduced in Section 4.3.1 can be seamlessly adjusted into this setting.
2. Later we show that given an S-GKAB, its transition system is E-bisimilar to the transition system of its corresponding S-CSGKAB that is obtained through τ_{sc} .
 3. Thus, utilizing Lemma 4.22 (except that now we consider a KB transition system and a context-sensitive transition system) and also by considering that $\mu\mathcal{L}_{\text{CTX}}$ without context expression is the same as $\mu\mathcal{L}_A^{\text{EQL}}$, we can easily recast the verification of S-GKABs into S-CSGKABs.

In the following two lemmas we aim to show that given an S-GKAB, its transition system is E-bisimilar to the transition system of its corresponding S-CSGKAB that is obtained through τ_{sc} .

Lemma 6.55. *Let \mathcal{G} be an S-GKAB with transition system $\mathcal{Y}_{\mathcal{G}}^{fs}$, and let $\tau_{sc}(\mathcal{G})$ be its corresponding S-CSGKAB (with transition system $\mathcal{Y}_{\tau_{sc}(\mathcal{G})}^{f_{S}^{cx}}$) obtained through τ_{sc} . Consider a state $s_s = \langle A_s, m_s, \delta_s \rangle$ of $\mathcal{Y}_{\mathcal{G}}^{fs}$, and a state $s_{cx} = \langle A_{cx}, m_{cx}, C, \delta_{cx} \rangle$ of $\mathcal{Y}_{\tau_{sc}(\mathcal{G})}^{f_{S}^{cx}}$. If $A_{cx} = A_s$, $m_{cx} = m_s$, and $\delta_{cx} = \kappa_{sc}(\delta_s)$, then $\langle A_{cx}, m_{cx}, C, \delta_{cx} \rangle \sim_E \langle A_s, m_s, \delta_s \rangle$.*

Proof. Now, let

1. $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ and $\mathcal{Y}_{\mathcal{G}}^{fs} = \langle \Delta, T, \Sigma_s, s_{0s}, abox_s, \Rightarrow_s \rangle$.
2. $\tau_{sc}(\mathcal{G}) = \langle T_{cx}, A_0, \Gamma, \delta', C_0, \Pi_C \rangle$, and $\mathcal{Y}_{\tau_{sc}(\mathcal{G})}^{f_{S}^{cx}} = \langle \Delta, T_{cx}, \Sigma_{cx}, s_{0cx}, abox_{cx}, ctx, \Rightarrow_{cx} \rangle$,

Now, we have to show the following: For every state $\langle A'_s, m'_s, \delta'_s \rangle$ such that $\langle A_s, m_s, \delta_s \rangle \Rightarrow_s \langle A'_s, m'_s, \delta'_s \rangle$ there exists $\langle A'_{cx}, m'_{cx}, C', \delta'_{cx} \rangle$ such that

- (a) we have $\langle A_{cx}, m_{cx}, C, \delta_{cx} \rangle \Rightarrow \langle A'_{cx}, m'_{cx}, C', \delta'_{cx} \rangle$,
- (b) $A'_s = A'_{cx}$
- (c) $m'_s = m'_{cx}$;
- (d) $\delta'_{cx} = \kappa_{sc}(\delta'_s)$.

The proof can be easily obtained by considering that within S-CSGKAB, by the definition of τ_{sc} (see Definition 6.54), it is easy to see that the following hold:

- There is only one possible context that is $C = \{[d \rightsquigarrow \top_d]\}$.
- The initial context is $C_0 = \{[d \rightsquigarrow \top_d]\}$ and it stays the same along the system evolution because we only have a single context evolution rule $\langle \text{true}, [d \rightsquigarrow \top_d] \rangle \mapsto \{[d \rightsquigarrow \top_d]\}$ that essentially never change the context. Thus, the TBox stay the same for all states. Moreover, all of the TBox assertions hold in our only one possible context since for each $\langle t : \varphi \rangle \in T_{cx}$ we have $\varphi = [d \rightsquigarrow \top_d]$. Therefore, basically the situation of the TBox is the same as in the original S-GKAB.
- Each context-sensitive action invocation in δ' has a context expression $\varphi = [d \rightsquigarrow \top_d]$. Thus, basically we can ignore it since it will always satisfied due to all of the facts above. Due to this fact, each context-sensitive action invocation in δ' is the same as the usual action invocation in δ .
- By the definition of execution semantics of S-CSGKABs and S-GKAB, they have the same way in updating an ABox and both of them reject each action execution that leads into an inconsistent state.

□

Lemma 6.56. *Given an S-GKAB \mathcal{G} , we have $\Upsilon_{\mathcal{G}}^{fs} \sim_E \Upsilon_{\tau_{sc}(\mathcal{G})}^{f_{sc}^{cx}}$*

Proof. Let

1. $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ and $\Upsilon_{\mathcal{G}}^{fs} = \langle \Delta, T, \Sigma_s, s_{0s}, abox_s, \Rightarrow_s \rangle$.
2. $\tau_{sc}(\mathcal{G}) = \langle T_{cx}, A_0, \Gamma, \delta', C_0, \Pi_C \rangle$, and
 $\Upsilon_{\tau_{sc}(\mathcal{G})}^{f_{sc}^{cx}} = \langle \Delta, T_{cx}, \Sigma_{cx}, s_{0cx}, abox_{cx}, ctx, \Rightarrow_{cx} \rangle$,

We have that $s_{0s} = \langle A_0, m_s, \delta \rangle$ and $s_{0cx} = \langle A_0, m_{cx}, C_0, \delta' \rangle$ where $m_s = m_{cx} = \emptyset$. By the definition of κ_{sc} and τ_{sc} , we also have that their initial ABoxes are the same, and $\delta' = \kappa_{sc}(\delta)$. Hence, by Lemma 6.55, we have $s_{0s} \sim_E s_{0cx}$. Therefore, by the definition of E-bisimulation, we have $\Upsilon_{\mathcal{G}}^{fs} \sim_E \Upsilon_{\tau_{sc}(\mathcal{G})}^{f_{sc}^{cx}}$. \square

Having all machinery in hand, we now show that the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ properties over S-GKAB can be recast as verification over S-CSGKAB as follows.

Theorem 6.57. *Verification of closed $\mu\mathcal{L}_A^{\text{EQL}}$ properties over S-GKABs can be recast as verification over S-CSGKABs.*

Proof. By Lemma 6.56, we have that $\Upsilon_{\mathcal{G}}^{fs} \sim_E \Upsilon_{\tau_{sc}(\mathcal{G})}^{f_{sc}^{cx}}$. Hence, by Lemma 4.22 (but consider that now it is between a KB transition system and a context-sensitive transition system), for every $\mu\mathcal{L}_A^{\text{EQL}}$ property Φ , we have that

$$\Upsilon_{\mathcal{G}}^{fs} \models \Phi \text{ if and only if } \Upsilon_{\tau_{sc}(\mathcal{G})}^{f_{sc}^{cx}} \models \Phi$$

Hence, by using the translation τ_{sc} we can easily transform an S-GKAB into an S-CSGKAB and then the claim is easily follows due to the fact above. \square

6.7 Discussion

In Section 6.3 we have seen how we incorporate contextual information as an additional information that influence the flow of program execution. In particular, we lift the usual atomic action invocations into context-sensitive atomic action invocations that are not only guarded by queries but also with context expressions (see Definition 6.12). Now, observe that we can actually also extend the conditional and loop constructs (i.e., **if** φ **then** δ_1 **else** δ_2 and **while** φ **do** δ) such that they also incorporate contextual information. Formally, we can easily augment context expressions as an additional “if condition” (resp. “loop guard”) as follows:

$$\begin{aligned} &\textbf{if } (\varphi, \varphi_C) \textbf{ then } \delta_1 \textbf{ else } \delta_2 \\ &\textbf{while } (\varphi, \varphi_C) \textbf{ do } \delta \end{aligned}$$

Furthermore we can also easily adjust the execution semantics concerning the two constructs such that they are context-sensitive. For the conditional construct **if** (φ, φ_C) **then** δ_1 **else** δ_2 , we can require that δ_1 is executable in case φ is successfully evaluated over the current KB and φ_C is entailed by the corresponding current context together with the corresponding value domain theory. Similarly, for the loop construct **while** (φ, φ_C) **do** δ , we can require that the program δ will be executed as long as φ is successfully evaluated over the current KB and φ_C is entailed by the corresponding current context together with the corresponding value domain theory.

Interestingly, with the extensions above, we can still reduce the verification of S-CSGKABs into the corresponding verification of S-GKABs. This can be done easily since we can emulate the context expression as a query. Thus we can follow the similar way of simulating context-sensitive atomic action invocation inside S-GKABs. Another interesting point is that we can also easily simulate S-GKABs inside S-CSGKABs. Similar approach as in Section 6.6 can be followed.

One might also extend the framework further by introducing a construct that contextualized a program in general. I.e., introduce the following construct:

$$\varphi_C : \delta$$

There are actually two possible ways in defining the semantics of such construct:

1. The first semantics would be constraining that the whole program δ can only be executed as long as the current context C together with the value domain theory Φ_D entail φ_C .
2. The second semantics would be constraining that we can start to execute the program δ if the current context C together with the value domain theory Φ_D entail φ_C .

The different between those two semantics is that in the second semantics, the checking whether the context expression φ_C holds or not is only done when we want to start to execute the program δ , while in the first semantics, the checking whether the context expression φ_C holds or not is done along the execution of δ (i.e., each atomic action invocation within δ can only be executed if the context expression φ_C holds).

More formally, to adopt the first semantics, we can extend the definition of context-sensitive program execution relation (cf. Definition 6.21) by adding the following:

- $\langle A, m, C, \varphi_C : \delta \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A', m', C', \varphi_C : \delta' \rangle$, if
 $\langle A, m, C, \delta \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A', m', C', \delta' \rangle$, and $C \cup \Phi_D \models \varphi_C$.

On the other hand to adopt the second semantics, we can extend the definition of context-sensitive program execution relation (cf. Definition 6.21) by adding the following:

- $\langle A, m, C, \varphi_C : \delta \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A', m', C', \delta' \rangle$, if
 $\langle A, m, C, \delta \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A', m', C', \delta' \rangle$, and $C \cup \Phi_D \models \varphi_C$.

Notice that no matter whether we adopt the first or the second semantics, it can still be shown that verification of S-CSGKABs can be reduced to verification of S-GKABs. For the first semantics, notice that a program expression $\varphi_C : \delta$ can be translated into the standard contextualized Golog program introduced before (cf. Definition 6.12) by “distributing” the context expressions φ_C into each atomic action invocation in δ . By doing this, each atomic action invocation will be constrained by the context expressions φ_C . For the second semantics, we can emulate the construct $\varphi_C : \delta$ by using the contextualized “if” construct introduced above. Essentially, having $\varphi_C : \delta$ is the same as having

$$\text{if } (\text{true}, \varphi_C) \text{ then } \delta \text{ else pick false.}\alpha().$$

INCONSISTENCY-AWARE CONTEXT-SENSITIVE GKABs

In Chapter 5 we have seen how GKABs can be lifted into inconsistency-aware GKABs such that they handle inconsistency in a more sophisticated way. Moreover, in Chapter 6 we have also seen an extension of GKABs into context-sensitive GKABs which taking into account the contextual information during the evolution of the system. Now, in this chapter we blend those two extensions and introduce the so called Inconsistency-aware Context-sensitive GKABs (I-CSGKABs).

Technically, to obtain I-CSGKABs, we start from CSGKABs (that has been introduced in Chapter 6) and then lift it into inconsistency-aware CSGKABs using a similar approach as the way how we get into inconsistency-aware GKABs in Chapter 5. I.e., we exploit the parametric execution semantics of CSGKABs and obtain inconsistency-aware CSGKABs by introducing various kind of filters that incorporate the inconsistency handling mechanisms and simply plug them in into CSGKABs.

In this chapter, we also tackle the problem of verifying $\mu\mathcal{L}_{\text{CTX}}$ properties over I-CSGKABs. Similar to how we deal with the verification problem of I-GKABs, in this chapter we show how we can reduce the verification of I-CSGKABs into S-GKABs. However, due to the presence of context, the TBox might change depending on the context. Therefore, the repair of the inconsistency should take into account the context and it must be performed based on the TBox assertions that “hold” within the corresponding context. Hence, when it comes to transforming I-CSGKABs into S-GKABs, we can not simply directly re-use all of the tools that we use to reduce the verification of I-GKABs into the verification of S-GKABs. The repair program (that simulates the repair computation) need to work based on the context. Thus, the challenge is how to make the repair program context-sensitive such that it performs the repair w.r.t. the TBox under the new context. I.e., the repair program (might) always need to be adjusted “on the fly” based on the context.

Similar to KABs and GKABs, in the following we use $DL\text{-}Lite_A$ for expressing KBs and we also do not distinguish between objects and values (thus we drop attributes). Moreover we make use of a countably infinite set Δ of constants, which intuitively denotes all possible values in the system. Additionally, we consider a finite set of distinguished constants $\Delta_0 \subset \Delta$, and a finite set \mathcal{F} of *function symbols* that represents *service calls*, which abstractly account for the injection of fresh values (constants) from Δ into the system. Additionally, for technical development of this chapter, except for Section 7.3, we fix a set $\mathbb{D} = \{d_1, \dots, d_n\}$ of context dimensions. Each context dimension $d_i \in \mathbb{D}$ has its own tree-shaped finite value domain $\langle \text{Dom}(d_i), \prec_{d_i} \rangle$, where $\text{Dom}(d_i)$ represents the finite set of domain values, and \prec_{d_i} represents the predecessor relation forming the tree.

7.1 The Inconsistency-aware Context-sensitive Execution Semantics

Similar to Section 5.2, the inconsistency-aware context-sensitive execution semantics for CSGKABs are obtained by simply exploiting the context-sensitive filter relations (in Definition 6.18) to define three inconsistency-aware semantics that incorporate the repair-based approaches reviewed in Section 5.1. In particular, we introduce three context sensitive filter relations namely f_B^{cx} , f_C^{cx} , and f_E^{cx} . For brevity, from this moment we often simply say filter to refer to context-sensitive filter relation.

As the first one, we define B-repair Context-sensitive Filter f_B^{cx} as follows:

*B-repair
Context-sensitive
Filter f_B^{cx}*

Definition 7.1 (B-repair Context-sensitive Filter f_B^{cx}). A *B-repair Context-sensitive Filter* f_B^{cx} is a relation that consists of tuples of the form $\langle A, F^+, F^-, C, A' \rangle$ such that $A' \in \text{B-REP}(T_{cx}^C, (A \setminus F^-) \cup F^+)$, where A and A' are ABoxes, C is a context, and F^+ as well as F^- are two sets of ABox assertions. ■

Employing the b-repair context-sensitive filter f_B^{cx} into CSGKABs gives us B-CSGKABs that is a CSGKABs with *b-repair execution semantics*, i.e., where inconsistent ABoxes are repaired by non-deterministically picking a b-repair. Formally, we define the transition system which provide the b-repair execution semantics for CSGKABs as follows.

*CSGKAB
B-Transition System*

Definition 7.2 (CSGKAB B-Transition System). Given a CSGKAB \mathcal{G}_{cx} and a b-repair context sensitive filter f_B^{cx} , the *b-transition system* of \mathcal{G}_{cx} , written $\mathcal{T}_{\mathcal{G}_{cx}}^{f_B^{cx}}$, is the transition system of \mathcal{G}_{cx} w.r.t. f_B^{cx} (see also Definition 6.22). ■

We call *B-CSGKABs* the CSGKABs adopting this semantics.

Example 7.3. Let the CSGKAB \mathcal{G}_{cx} specified in Example 6.14 be a B-CSGKAB. Consider the state $s = \langle A, m, C, \delta \rangle$ where:

- $A = \{ \text{ReceivedOrder}(\text{chair}), \text{ApprovedOrder}(\text{table}), \text{designedBy}(\text{table}, \text{alice}), \text{Designer}(\text{alice}), \text{hasDesign}(\text{table}, \text{ecodesign}), \text{hasAssemblingLoc}(\text{table}, \text{bolzano}) \}$,
- $m = \{ [\text{GETDESIGNER}(\text{table}) \rightarrow \text{alice}], [\text{GETDESIGN}(\text{table}) \rightarrow \text{ecodesign}], [\text{ASSIGNASSEMBLINGLOC}(\text{table}) \rightarrow \text{bolzano}] \}$,
- $C = \{ [\text{PP} \rightsquigarrow \text{WE}], [\text{S} \rightsquigarrow \text{PS}] \}$,
- $\delta = \delta_3; \delta_4; \delta_5$; **while** $\exists x. [\text{Order}(x)] \wedge \neg [\text{DeliveredOrder}(x)]$ **do** δ_0 .

Note that the state s is a reachable state from the initial state s_0 in the transition system $\mathcal{T}_{\mathcal{G}_{cx}}^{f_B^{cx}}$ of \mathcal{G}_{cx} . One possible successor state of s is a state $s_1 = \langle A_1, m_1, C_1, \delta_1 \rangle$ with

- $A_1 = \{ \text{ReceivedOrder}(\text{chair}), \text{designedBy}(\text{table}, \text{alice}), \text{Designer}(\text{alice}), \text{hasDesign}(\text{table}, \text{ecodesign}), \text{hasAssemblingLoc}(\text{table}, \text{bolzano}), \text{AssembledOrder}(\text{table}), \text{assembledBy}(\text{table}, \text{alice}) \}$,

- $m_1 = \{ \text{[GETDESIGNER(table)} \rightarrow \text{alice}], \text{[GETDESIGN(table)} \rightarrow \text{ecodesign}], \text{[ASSIGNASSEMBLINGLOC(table)} \rightarrow \text{bolzano}], \text{[GETASSEMBLER(table)} \rightarrow \text{alice}], \text{[GETASSEMBLINGLOC(table)} \rightarrow \text{bolzano}] \}$,
- $C_1 = \{[\text{PP} \rightsquigarrow \text{N}], [\text{S} \rightsquigarrow \text{NS}]\}$,
- $\delta_1 = \delta_4; \delta_5; \text{while } \exists x. [\text{Order}(x)] \wedge \neg [\text{DeliveredOrder}(x)] \text{ do } \delta_0.$

The state s_1 is obtained from the execution of action invocation

pick $\langle \text{true}, [\text{PP} \rightsquigarrow \text{AP}] \wedge [\text{S} \rightsquigarrow \text{AS}] \rangle. \text{assembleOrders}()$

where the context is changing from C to C' due to the application of the following context evolution rule:

$$\langle \text{true}, [\text{PP} \rightsquigarrow \text{RE}] \wedge [\text{S} \rightsquigarrow \text{PS}] \rangle \mapsto \{[\text{PP} \rightsquigarrow \text{N}], [\text{S} \rightsquigarrow \text{NS}]\}.$$

Furthermore, we have that

$$A_1 \in \text{B-REP}(T_{cx}^{C_1}, (A \setminus \{\text{ApprovedOrder}(\text{table})\}) \cup \{\text{AssembledOrder}(\text{table}), \text{assembledBy}(\text{table}, \text{alice}), \text{Assembler}(\text{alice}), \text{hasAssemblingLoc}(\text{table}, \text{bolzano})\})$$

To obtain c-repair execution semantics, now we proceed to define the c-repair context-sensitive filter as follows.

Definition 7.4 (C-repair Context-sensitive Filter f_C^{cx}). A *C-repair Context-sensitive Filter* f_C^{cx} is a relation that consists of tuples of the form $\langle A, F^+, F^-, C, A' \rangle$ such that $A' = \text{C-REP}(T_{cx}^C, (A \setminus F^-) \cup F^+)$, where A and A' are ABoxes, C is a context, and F^+ as well as F^- are two sets of ABox assertions. ■

*C-repair
Context-sensitive
Filter f_C^{cx}*

Filter f_C^{cx} gives rise to the *c-repair execution semantics* for CSGKABs, where inconsistent ABoxes are repaired by computing their unique c-repair. The transition systems which provide the c-repair execution semantics for CSGKABs is then defined as follows.

Definition 7.5 (CSGKAB C-Transition System). Given a CSGKAB \mathcal{G}_{cx} and a c-repair filter f_C^{cx} , the *c-transition system* of \mathcal{G}_{cx} , written $\Upsilon_{\mathcal{G}_{cx}}^{f_C^{cx}}$, is the transition system of \mathcal{G}_{cx} w.r.t. f_C^{cx} (see also Definition 6.22). ■

*CSGKAB
C-Transition System*

We call *C-CSGKABs* the CSGKABs adopting this semantics.

Example 7.6. Let the CSGKAB \mathcal{G}_{cx} specified in Example 6.14 be a C-CSGKAB. Consider the state $s = \langle A, m, C, \delta \rangle$ where:

- $A = \{ \text{ReceivedOrder}(\text{chair}), \text{ApprovedOrder}(\text{table}), \text{designedBy}(\text{table}, \text{alice}), \text{Designer}(\text{alice}), \text{hasDesign}(\text{table}, \text{ecodesign}), \text{hasAssemblingLoc}(\text{table}, \text{bolzano}) \}$,
- $m = \{ [\text{GETDESIGNER}(\text{table}) \rightarrow \text{alice}], [\text{GETDESIGN}(\text{table}) \rightarrow \text{ecodesign}], [\text{ASSIGNASSEMBLINGLOC}(\text{table}) \rightarrow \text{bolzano}] \}$,
- $C = \{ [\text{PP} \rightsquigarrow \text{WE}], [\text{S} \rightsquigarrow \text{PS}] \}$,
- $\delta = \delta_3; \delta_4; \delta_5; \text{while } \exists x. [\text{Order}(x)] \wedge \neg [\text{DeliveredOrder}(x)] \text{ do } \delta_0.$

Note that the state s is a reachable state from the initial state s_0 in the transition system $\Upsilon_{\mathcal{G}_{cx}}^{f_C^{cx}}$ of \mathcal{G}_{cx} . One possible successor state of s is a state $s_1 = \langle A_1, m_1, C_1, \delta_1 \rangle$ with

- $A_1 = \{ \text{ReceivedOrder}(\text{chair}), \text{designedBy}(\text{table}, \text{alice}), \text{hasDesign}(\text{table}, \text{ecodesign}), \text{AssembledOrder}(\text{table}), \text{assembledBy}(\text{table}, \text{alice}) \}$,
- $m_1 = \{ [\text{GETDESIGNER}(\text{table}) \rightarrow \text{alice}], [\text{GETDESIGN}(\text{table}) \rightarrow \text{ecodesign}], [\text{ASSIGNASSEMBLINGLOC}(\text{table}) \rightarrow \text{bolzano}], [\text{GETASSEMBLER}(\text{table}) \rightarrow \text{alice}], [\text{GETASSEMBLINGLOC}(\text{table}) \rightarrow \text{trento}] \}$,
- $C_1 = \{ [\text{PP} \rightsquigarrow \text{N}], [\text{S} \rightsquigarrow \text{NS}] \}$,
- $\delta_1 = \delta_4; \delta_5; \text{while } \exists x. [\text{Order}(x)] \wedge \neg [\text{DeliveredOrder}(x)] \text{ do } \delta_0.$

The state s_1 is obtained from the execution of action invocation

pick $\langle \text{true}, [\text{PP} \rightsquigarrow \text{AP}] \wedge [\text{S} \rightsquigarrow \text{AS}] \rangle. \text{assembleOrders}()$

where the context is changing from C to C' due to the application of the following context evolution rule:

$$\langle \text{true}, [\text{PP} \rightsquigarrow \text{RE}] \wedge [\text{S} \rightsquigarrow \text{PS}] \rangle \mapsto \{ [\text{PP} \rightsquigarrow \text{N}], [\text{S} \rightsquigarrow \text{NS}] \}.$$

Furthermore, we have that

$$A_1 = \text{C-REP}(T_{cx}^{C_1}, (A \setminus \{ \text{ApprovedOrder}(\text{table}) \}) \cup \{ \text{AssembledOrder}(\text{table}), \text{assembledBy}(\text{table}, \text{alice}), \text{Assembler}(\text{alice}), \text{hasAssemblingLoc}(\text{table}, \text{trento}) \})$$

Next, we define the evolution filter which handle the inconsistency using the bold-evolution mechanism as in Definition 5.4.

Definition 7.7 (B-evol Context-sensitive Filter f_E^{cx}). A *B-evol Context-sensitive Filter* f_E^{cx} is a relation that consists of tuples of the form $\langle A, F^+, F^-, C, A' \rangle$ such that $A' = \text{EVOL}(T_{cx}^C, A, F^+, F^-)$, and F^+ is T_{cx}^C -consistent, where A and A' are ABoxes, C is a context, and F^+ as well as F^- are two sets of ABox assertions. ■

*B-evol
Context-sensitive
Filter f_E^{cx}*

Filter f_E^{cx} gives rise to the *b-evol execution semantics* for CSGKABs, where for updates leading to inconsistent ABoxes, their unique bold-evolution is computed. Notice that by combining the definition of CS-TELL (see Definition 6.19) and filter f_E^{cx} , we basically assume that the new ABox assertions are consistent with the TBox under the new context (i.e., after the context change). This means that we assume that the update is always accepted/applicable since it gives the system the new information. The transition systems which provide the b-evol execution semantics for CSGKABs is defined as follows.

Definition 7.8 (CSGKAB E-Transition System). Given a CSGKAB \mathcal{G}_{cx} and a e-repair filter f_E^{cx} , the *e-transition system* of \mathcal{G}_{cx} , written $\Upsilon_{\mathcal{G}_{cx}}^{f_E^{cx}}$, is the transition system of \mathcal{G}_{cx} w.r.t. f_E^{cx} . ■

*CSGKAB
E-Transition System*

We call *E-CSGKABs* the CSGKABs adopting this semantics. We group these three forms of CSGKABs (i.e., B-CSGKABs, C-CSGKABs, E-CSGKABs) under the umbrella of *inconsistency-aware CSGKABs (I-CSGKABs)*. The definition of $\mu\mathcal{L}_{\text{CTX}}$ verification over I-CSGKABs is as usual, i.e., similar to the case of CSGKABs (see Definition 6.27).

Example 7.9. Let the CSGKAB \mathcal{G}_{cx} specified in Example 6.14 be an E-CSGKAB. Consider the state $s = \langle A, m, C, \delta \rangle$ where:

- $A = \{ \text{ReceivedOrder}(\text{chair}), \text{ApprovedOrder}(\text{table}), \text{designedBy}(\text{table}, \text{alice}), \text{Designer}(\text{alice}), \text{hasDesign}(\text{table}, \text{ecodesign}), \text{hasAssemblingLoc}(\text{table}, \text{bolzano}) \},$
- $m = \{ [\text{GETDESIGNER}(\text{table}) \rightarrow \text{alice}], [\text{GETDESIGN}(\text{table}) \rightarrow \text{ecodesign}], [\text{ASSIGNASSEMBLINGLOC}(\text{table}) \rightarrow \text{bolzano}] \},$
- $C = \{ [\text{PP} \rightsquigarrow \text{WE}], [\text{S} \rightsquigarrow \text{PS}] \},$
- $\delta = \delta_3; \delta_4; \delta_5; \text{while } \exists x. [\text{Order}(x)] \wedge \neg [\text{DeliveredOrder}(x)] \text{ do } \delta_0.$

Note that the state s is a reachable state from the initial state s_0 in the transition system $\Upsilon_{\mathcal{G}_{cx}}^{f_E^{cx}}$ of \mathcal{G}_{cx} . One possible successor state of s is a state $s_1 = \langle A_1, m_1, C_1, \delta_1 \rangle$ with

- $A_1 = \{ \text{ReceivedOrder}(\text{chair}), \text{designedBy}(\text{table}, \text{alice}), \text{hasDesign}(\text{table}, \text{ecodesign}), \text{AssembledOrder}(\text{table}), \text{assembledBy}(\text{table}, \text{alice}), \text{Assembler}(\text{alice}), \text{hasAssemblingLoc}(\text{table}, \text{trento}) \},$

- $m_1 = \{ \text{[GETDESIGN(table)} \rightarrow \text{alice}], \text{[GETDESIGN(table)} \rightarrow \text{ecodesign}],$
 $\text{[ASSIGNASSEMBLINGLOC(table)} \rightarrow \text{bolzano}],$
 $\text{[GETASSEMBLER(table)} \rightarrow \text{alice}],$
 $\text{[GETASSEMBLINGLOC(table)} \rightarrow \text{trento}] \}$,
- $C_1 = \{[\text{PP} \rightsquigarrow \text{N}], [\text{S} \rightsquigarrow \text{NS}]\}$,
- $\delta_1 = \delta_4; \delta_5; \text{while } \exists x. [\text{Order}(x)] \wedge \neg [\text{DeliveredOrder}(x)] \text{ do } \delta_0.$

The state s_1 is obtained from the execution of action invocation

pick $\langle \text{true}, [\text{PP} \rightsquigarrow \text{AP}] \wedge [\text{S} \rightsquigarrow \text{AS}] \rangle. \text{assembleOrders}()$

where the context is changing from C to C' due to the application of the following context evolution rule:

$$\langle \text{true}, [\text{PP} \rightsquigarrow \text{RE}] \wedge [\text{S} \rightsquigarrow \text{PS}] \rangle \mapsto \{[\text{PP} \rightsquigarrow \text{N}], [\text{S} \rightsquigarrow \text{NS}]\}.$$

Furthermore, we have that $A_1 = \text{EVOL}(T_{cx}^{C_1}, A, F^+, F^-)$ where F^+ and F^- are the set of assertions to be added and deleted by the action **assembleOrders/0** as follows:

$$F^+ = \{ \text{AssembledOrder}(\text{table}), \text{assembledBy}(\text{table}, \text{alice}), \\ \text{Assembler}(\text{alice}), \text{hasAssemblingLoc}(\text{table}, \text{trento}) \}$$

$$F^- = \{ \text{ApprovedOrder}(\text{table}) \}$$

7.2 From Inconsistency-aware Context-sensitive GKABS to Standard GKABS

In this section we show that all I-CSGKABS introduced in Section 7.1 (i.e., B-CSGKABS, C-CSGKABS, and E-CSGKABS) can be compiled into S-GKABS. In particular, we show that verification of $\mu\mathcal{L}_{\text{CTX}}$ formulas over I-CSGKABS can be reduced to the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ over S-GKABS. To this aim, here we combine our results in Chapter 5 and Chapter 6.

For a technical reason, we reserve a special ABox assertion **State**(*temp*), where *temp* $\in \Delta_0$ and **State** is a reserved concept name (i.e., outside of any TBox vocabulary). Basically, we use **State**(*temp*) to distinguish *stable* states, where an atomic action can be applied, from intermediate states used by the S-GKABS to mimics the context evolution as well as (incrementally) remove inconsistent assertions from the ABox. Stable/intermediate states are marked by the absence/presence of **State**(*temp*). As before, here the ABox assertion **State**(*temp*) is often also called special marker.

Similar to Section 6.5.2.1, since the changes of context requires the original ABox and we do it after the action execution, we do not materialize the result of an action execution directly after its execution, instead we just mark the assertions that should

be added/deleted, and concretize it during the execution of the action that evolves context. To this aim, for each concept name $N \in \text{voc}(T_{cx})$, we introduce two fresh concept name N^a and N^d to keep track the temporary information about ABox assertions to be added/deleted before we materialize the update (similarly for roles). Consecutively, we call such kind of concept names *added* and *deleted fact marker* concept names. Additionally, when we compile E-CSGKABs into S-GKABs, we also use added fact marker concept names to mark the information about newly added assertions.

In order to mimic the context evolution within S-GKABs, we simply adopt our approach in Section 6.5.2.1. Thus, similar to Section 6.5.2.1, for each context dimension assignment $[d_i \rightsquigarrow v_j]$ we reserve two fresh concept names $D_i^{v_j}$ and $\mathbf{D}_i^{v_j}$ in order to represent it as an ABox assertion. Similarly, this kind of concept name is also called *context dimension concept*.

The following sections is then organized as follows: First we show how we compile B-CSGKABs into S-GKABs and show that the verification of $\mu\mathcal{L}_{\text{CTX}}$ over B-CSGKABs can be reduced into the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ over S-GKABs. After that, we also show similar results for C-CSGKABs and E-CSGKABs. Last, we close the tour by exhibiting our core result that the verification of I-CSGKABs can be reduced into the verification of S-GKABs.

7.2.1 From B-CSGKABs into Standard GKABs

We start this section by exhibiting how we translate B-CSGKABs into S-GKABs such that the resulting S-GKAB simulates the evolution of the given B-CSGKAB and also we will show how we translate the $\mu\mathcal{L}_{\text{CTX}}$ formulas into $\mu\mathcal{L}_A^{\text{EQL}}$ formulas with the aim to reduce the verification of B-CSGKABs into S-GKABs. While defining the translation from B-CSGKABs into S-GKABs, we also introduce the notion of context-sensitive b-repair program that is used to simulate the b-repair computation inside S-GKABs. After that, we continue to show the termination and the correctness of context-sensitive b-repair program by lifting the results in Section 5.3.1.1. The journey is then continued by introducing a certain bisimulation relation that will be used to prove the reduction of the $\mu\mathcal{L}_{\text{CTX}}$ verification over B-CSGKABs into the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ over S-GKABs. Last, we close this section by presenting the proof that we can recast the verification of B-CSGKABs into the corresponding verification of S-GKABs.

7.2.1.1 Translating B-CSGKABs into S-GKABs

Essentially, a single transition in the transition system of B-CSGKABs do the following:

1. Update the ABox based on the executed action.
2. Change the context.
3. Apply the b-repair to the newly updated ABox. Additonally, the b-repair is applied w.r.t. the TBox under the new context.

Therefore, in order to mimic the evolution of B-CSGKABs inside S-GKABs, we do the following:

1. First, to simulate the ABox and context evolution, we adopt the way how we simulate the ABox and context evolution of S-CSGKABs inside S-GKABs as in Section 6.5.2.1 but dropping the inconsistency check,
2. Then, to simulate the b-repair computation, we adopt the way how we simulate b-repair computation when we compile B-GKABs into S-GKABs (see Section 5.3.1). However, we can not use the b-repair program as in Definition 5.20 directly, because the TBox is changing depending on the context. Thus, the challenge is how to make the b-repair program context-sensitive such that it is always do the b-repair w.r.t. the TBox under the new context. I.e., the b-repair program (might) always need to be adjusted “on the fly” based on the current context.
3. Last, combining the two steps above sequentially gives us an S-GKAB that mimics the evolution of the given B-CSGKAB.

As a preliminary towards defining the translation from B-CSGKABs to S-KAB, we first define the notion of context-sensitive b-repair actions and context-sensitive b-repair atomic action invocations which will be used to define a context-sensitive b-repair program. The main purpose of introducing the context-sensitive b-repair program is to mimic the computation of b-repair in B-CSGKABs while also taking into account the context, and thus we can mimic the whole computation in B-CSGKABs inside S-GKAB.

Definition 7.10 (Context-sensitive B-Repair Actions and Atomic Action Invocations). Given a Contextualized TBox T_{cx} , let $\text{CTX}(\mathbb{D})$ be the set of all possible context (see Definition 6.28). We define the set $\Gamma_b^{T_{cx}}$ of *b-repair actions* over T_{cx} and the set $\Lambda_b^{T_{cx}}$ of *b-repair atomic action invocations* over T_{cx} as follows: for each context $C \in \text{CTX}(\mathbb{D})$, we have:

1. For each functionality assertion $(\text{funct } R) \in T_{cx}^C$, we include in $\Gamma_b^{T_{cx}}$ and $\Lambda_b^{T_{cx}}$ respectively:
 - $\alpha_F(x, y) : \{R(x, z) \wedge \neg[z = y] \rightsquigarrow \mathbf{del} \{R(x, z)\}\} \in \Gamma_b^{T_{cx}}$, and
 - $\mathbf{pick} (q_C \wedge \exists z. q_{\text{unsat}}^f((\text{funct } R), x, y, z)).\alpha_F(x, y) \in \Lambda_b^{T_{cx}}$.

Essentially, the atomic action invocation and action above together repair an inconsistency related to $(\text{funct } R)$ by removing all tuples causing the inconsistency, except one.

2. For each negative concept inclusion $B_1 \sqsubseteq \neg B_2$ such that $T_{cx}^C \models B_1 \sqsubseteq \neg B_2$, we include in $\Gamma_b^{T_{cx}}$ and $\Lambda_b^{T_{cx}}$ respectively:
 - $\alpha_{B_1}(x) : \{\mathbf{true} \rightsquigarrow \mathbf{del} \{B_1(x)\}\} \in \Gamma_b^{T_{cx}}$, and
 - $\mathbf{pick} (q_C \wedge q_{\text{unsat}}^n(B_1 \sqsubseteq \neg B_2, x)).\alpha_{B_1}(x) \in \Lambda_b^{T_{cx}}$.

Basically, the atomic action invocation and action above together repair an inconsistency related to $B_1 \sqsubseteq \neg B_2$ by removing a constant that is both in B_1 and B_2 from B_1 .

3. For each negative role inclusion $R_1 \sqsubseteq \neg R_2$ such that $T_{cx}^C \models R_1 \sqsubseteq \neg R_2$, we include in $\Gamma_b^{T_{cx}}$ and $\Lambda_b^{T_{cx}}$ respectively:
 - $\alpha_{R_1}(x, y) : \{\mathbf{true} \rightsquigarrow \mathbf{del} \{R_1(x, y)\}\} \in \Gamma_b^{T_{cx}}$, and
 - $\mathbf{pick} (q_C \wedge q_{\text{unsat}}^n(R_1 \sqsubseteq \neg R_2, x, y)).\alpha_{R_1}(x, y) \in \Lambda_b^{T_{cx}}$.

The atomic action invocation and action above repair an inconsistency related to $R_1 \sqsubseteq \neg R_2$ by removing constants that is both in R_1 and R_2 from R_1 .

■

As the b-repair program in Section 5.3.1, we will see later that the context-sensitive b-repair program essentially will non-deterministically choose the context-sensitive b-repair atomic action invocations while there is still an inconsistency. Therefore, in order to specify the guard of the while loop (i.e., to check whether there is still an inconsistency), we need to define the notion of context-sensitive Q-UNSAT, by leveraging on the usual notion of Q-UNSAT, as follows.

Definition 7.11 (Context-sensitive Q-UNSAT-ECQ). Given a contextualized TBox T_{cx} , a *context-sensitive Q-UNSAT-ECQ* over T_{cx} is a boolean FOL query $Q_{\text{unsatECQ}}^{T_{cx}}$ of the following form:

*Context-sensitive
Q-UNSAT-ECQ*

$$Q_{\text{unsatECQ}}^{T_{cx}} = \left(\bigvee_{C \in \text{CTX}(\mathbb{D})} (q_C \wedge Q_{\text{unsatECQ}}^{T_{cx}^C}) \right)$$

where

- $Q_{\text{unsatECQ}}^{T_{cx}^C}$ is Q-UNSAT-ECQ over TBox T_{cx}^C (Note that T_{cx}^C is T_{cx} under the context C),
- $\text{CTX}(\mathbb{D})$ is the set of all possible context as in Definition 6.28,
- q_C is the query obtained from the context C .

■

We now proceed to define the context-sensitive b-repair program by utilizing the context-sensitive b-repair atomic action invocations as well as the context-sensitive b-repair actions that has been introduced before.

Definition 7.12 (Context-sensitive B-Repair Program). Given a B-CSGKABs $\mathcal{G}_{cx} = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$. Let $\Gamma_b^{T_{cx}}$ be a set of context-sensitive b-repair actions over T_{cx} , and $\Lambda_b^{T_{cx}} = \{a_1, \dots, a_n\}$ be a set of context-sensitive b-repair atomic action invocations over T_{cx} . We then define the *context-sensitive b-repair program* over T_{cx} as follows:

*Context-sensitive
B-Repair Program*

$$\delta_b^{T_{cx}} = \mathbf{while} \ Q_{\text{unsatECQ}}^{T_{cx}} \ \mathbf{do} \ \delta_r$$

where $\delta_r = a_1 | a_2 | \dots | a_n$.

■

As the last step before we formally define the translation of B-CSGKABs into S-GKABs, below we define the program translation that will be used to translate the program in B-CSGKABs. Basically we define a translation function κ_B^{cx} that concatenates each action invocation with a program that non-deterministically choose an action that changes the context, and then concatenates them with the context-sensitive b-repair program. Additionally, the translation function κ_B^{cx} also serves as a one-to-one correspondence (bijection) between the original and the translated program (as well as between the sub-program).

Program Translation
 κ_B^{cx}

Definition 7.13 (Program Translation κ_B^{cx}). Given a B-CSGKABs $\mathcal{G}_{cx} = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$ we define a *translation* κ_B^{cx} which translates a program δ into a program δ' inductively as follows:

$$\begin{aligned} \kappa_B^{cx}(\mathbf{pick} \langle Q(\vec{p}), \varphi_C \rangle. \alpha(\vec{p})) &= \mathbf{pick} Q'(\vec{p}). \alpha'(\vec{p}); \delta_{\Pi_C}; \delta_b^{T_{cx}}; \mathbf{pick} \text{true}. \alpha_{temp}^-() \\ \kappa_B^{cx}(\varepsilon) &= \varepsilon \\ \kappa_B^{cx}(\delta_1 | \delta_2) &= \kappa_B^{cx}(\delta_1) | \kappa_B^{cx}(\delta_2) \\ \kappa_B^{cx}(\delta_1; \delta_2) &= \kappa_B^{cx}(\delta_1); \kappa_B^{cx}(\delta_2) \\ \kappa_B^{cx}(\mathbf{if} \varphi \mathbf{then} \delta_1 \mathbf{else} \delta_2) &= \mathbf{if} \varphi \mathbf{then} \kappa_B^{cx}(\delta_1) \mathbf{else} \kappa_B^{cx}(\delta_2) \\ \kappa_B^{cx}(\mathbf{while} \varphi \mathbf{do} \delta) &= \mathbf{while} \varphi \mathbf{do} \kappa_B^{cx}(\delta) \end{aligned}$$

where

- $\mathbf{pick} Q'(\vec{p}). \alpha'(\vec{p})$ is an action invocation obtained from $\mathbf{pick} \langle Q(\vec{p}), \varphi_C \rangle. \alpha(\vec{p})$ as in Definition 6.37,
- δ_{Π_C} is a context-change program obtained from Π_C as in Definition 6.39,
- $\delta_b^{T_{cx}}$ is a context-sensitive b-repair program over T_{cx} as in Definition 7.12,
- $\alpha_{temp}^-() : \{\text{true} \rightsquigarrow \mathbf{del} \{\text{State}(temp)\}\}$.

■

Having all of the machinery in hand, we are ready to define a translation τ_B^{cx} that, given a B-CSGKAB, produces an S-GKAB as follows:

Translation from
B-CSGKAB to
S-GKAB

Definition 7.14 (Translation from B-CSGKAB to S-GKAB). We define a translation τ_B^{cx} that, given a B-CSGKAB $\mathcal{G}_{cx} = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$, produces an S-GKAB $\tau_B^{cx}(\mathcal{G}_{cx}) = \langle T_D, A_0 \cup A_{C_0}, \Gamma', \delta' \rangle$, where

- T_D is a TBox obtained from a set of context dimensions \mathbb{ID} (see Definition 6.29),
- A_{C_0} is an ABox obtained from C_0 (see Definition 6.31),
- $\Gamma' = \Gamma_\alpha \cup \Gamma_C \cup \Gamma_b^{T_{cx}} \cup \{\alpha_{temp}^-\}$ where:
 - Γ_α is obtained from Γ such that for each action $\alpha \in \Gamma$, we have $\alpha' \in \Gamma_\alpha$ where α' is a delayed action obtained from α (see Definition 6.36),
 - Γ_C is obtained from Π_C such that for each context-evolution rule $\langle Q, \varphi_C \rangle \mapsto C_{new}$ in Π_C , we have $\alpha_C \in \Gamma_C$ where α_C is an action obtained from the context-evolution rule $\langle Q, \varphi_C \rangle \mapsto C_{new}$ (see Definition 6.38),
 - $\Gamma_b^{T_{cx}}$ is the set of context-sensitive b-repair actions over T_{cx} (see Definition 7.10),
 - α_{temp}^- is an action of the form $\alpha_{temp}^-() : \{\text{true} \rightsquigarrow \mathbf{del} \{\text{State}(temp)\}\}$.
- $\delta' = \kappa_B^{cx}(\delta)$.

■

The $\mu\mathcal{L}_{\text{CTX}}$ property Φ over a B-CSGKAB \mathcal{G}_{cx} can then be recast as a corresponding property over an S-GKAB $\tau_B^{cx}(\mathcal{G}_{cx})$ using the following formula translation:

Translation t_j^{cx}

Definition 7.15 (Translation t_j^{cx}). We define a *translation* t_j^{cx} that transforms an arbitrary $\mu\mathcal{L}_{\text{CTX}}$ formula Φ (in NNF) into another $\mu\mathcal{L}_A^{\text{EQL}}$ formula Φ' inductively by recurring over the structure of Φ as follows:

- $t_j^{cx}(Q) = Q_{cx}$
- $t_j^{cx}(\varphi_C) = q_{\varphi_C}$
- $t_j^{cx}(\neg Q) = \neg Q_{cx}$
- $t_j^{cx}(\mathcal{Q}x.\Phi) = \mathcal{Q}x.t_j^{cx}(\Phi)$
- $t_j^{cx}(\Phi_1 \circ \Phi_2) = t_j^{cx}(\Phi_1) \circ t_j^{cx}(\Phi_2)$
- $t_j^{cx}(\odot Z.\Phi) = \odot Z.t_j^{cx}(\Phi)$
- $t_j^{cx}(\langle \neg \rangle \Phi) = \langle \neg \rangle \mu Z.((\text{State}(\text{temp}) \wedge \langle \neg \rangle Z) \vee (\neg \text{State}(\text{temp}) \wedge t_j^{cx}(\Phi)))$
- $t_j^{cx}([\neg] \Phi) = [\neg] \mu Z.((\text{State}(\text{temp}) \wedge [\neg] Z \wedge \langle \neg \rangle \top) \vee (\neg \text{State}(\text{temp}) \wedge t_j^{cx}(\Phi)))$

where:

- \circ is a binary operator ($\vee, \wedge, \rightarrow$, or \leftrightarrow),
- \odot is least (μ) or greatest (ν) fix-point operator,
- \mathcal{Q} is forall (\forall) or existential (\exists) quantifier.

■

Having those two translations in hand, we show it later that $\mathcal{R}_{\mathcal{G}_{cx}}^{f_{B}^{cx}} \models \Phi$ if and only if $\mathcal{R}_{\tau_B^{cx}(\mathcal{G}_{cx})}^{f_S} \models t_j^{cx}(\Phi)$ which consequently means that the verification of $\mu\mathcal{L}_{\text{CTX}}$ over B-CSGKABs can be reduced to the corresponding verification of $\mu\mathcal{L}_A^{\text{EQL}}$ over S-GKABs.

7.2.1.2 Termination and Correctness of Context-sensitive B-repair Program

In this section we aim to show the termination and correctness of context-sensitive b-repair program by essentially lifting the result in Section 5.3.1.1. First, we lift the result about the termination of b-repair program into the case of context-sensitive b-repair program as follows:

Lemma 7.16. *Let $\mathcal{G}_{cx} = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$ be a B-CSGKAB, $\tau_B^{cx}(\mathcal{G}_{cx})$ be an S-GKAB (with transition system $\mathcal{R}_{\tau_B^{cx}(\mathcal{G}_{cx})}^{f_S}$) obtained from \mathcal{G}_{cx} through τ_B^{cx} , and $\delta_b^{T_{cx}}$ be a context-sensitive b-repair program over T_{cx} . We have that $\delta_b^{T_{cx}}$ is always terminate. I.e., given a state $\langle A, m, \delta_b^{T_{cx}} \rangle$ of $\mathcal{R}_{\tau_B^{cx}(\mathcal{G}_{cx})}^{f_S}$, every program execution trace induced by $\delta_b^{T_{cx}}$ on $\langle A, m, \delta_b^{T_{cx}} \rangle$ w.r.t. filter f_S is terminating.*

Proof. Similar to the proof of Lemma 5.28 except that we need to accommodate the presence of context that is encoded as ABox assertions. All of the supporting lemmas to prove Lemma 5.28 can be also easily lifted to the context-sensitive case in order to support the proof of this lemma. The important argument in this proof is that each step of the program $\delta_b^{T_{cx}}$ always reduce the number of assertions that is participated in making the inconsistency. Thus, at some point when there is no more inconsistency, the loop will be exited. \square

Furthermore, we can also lift the correctness result of b-repair program into the case of context-sensitive b-repair program below. Essentially, we show that the context-sensitive b-repair program produces the same result as the result of b-repair over KB.

Theorem 7.17. Let $\mathcal{G}_{cx} = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$ be a B-CSGKAB, $\tau_B^{cx}(\mathcal{G}_{cx})$ be an S-GKAB (with transition system $\Upsilon_{\tau_B^{cx}(\mathcal{G}_{cx})}^{fs}$) obtained from \mathcal{G}_{cx} through τ_B^{cx} , and $\delta_b^{T_{cx}}$ be a context-sensitive b-repair program over T_{cx} . Consider an ABox A , a service call map m , and a context C . We have that $\text{RES}(A \cup A_C, m, \delta_b^{T_{cx}}) = \text{B-REP}(T_{cx}^C, A)$, where A_C is a set of ABox assertions representing the context C .

Proof. Similar to the proof of Theorem 5.32 by also observing the following:

- All supporting lemmas to prove Theorem 5.32 can be also easily lifted to the context-sensitive case in order to support the proof of this lemma.
- The set $\text{B-REP}(T_{cx}^C, A)$ of all b-repairs is computed w.r.t. the TBox T_{cx}^C (i.e., T_{cx} under the context C).
- The presense of context determines which b-repair action invocation in the context-sensitive b-repair program $\delta_b^{T_{cx}}$ that is executable and hence influence the execution flow of the program $\delta_b^{T_{cx}}$. Moreover, $\delta_b^{T_{cx}}$ is executed under the context C . Therefore, by construction of $\delta_b^{T_{cx}}$, the context-sensitive b-repair action invocations that are executed are only those that is related to the TBox assertion in T_{cx}^C . Thus, the repair is done based on the TBox T_{cx} under the context C .

□

7.2.1.3 Context-sensitive Jumping Bisimulation (CJ-Bisimulation)

In this section we introduce the notion of Context-sensitive Jumping Bisimulation (CJ-Bisimulation) by leveraging on the notion of Jumping Bisimulation as in (see Section 4.4.1) and the notion of Skip-two Bisimulation (see Section 6.5.2.2). Furthermore, here we also show some properties related to CJ-Bisimulation.

Definition 7.18 (Context-sensitive Jumping Bisimulation (CJ-Bisimulation)). Let $\Upsilon_1 = \langle \Delta, T_{cx}, \Sigma_1, s_{01}, abox_1, ctx, \Rightarrow_1 \rangle$ be a context-sensitive transition system, and $\Upsilon_2 = \langle \Delta, T, \Sigma_2, s_{02}, abox_2, \Rightarrow_2 \rangle$ be a KB transition system, with $\text{ADOM}(abox_1(s_{01})) \subseteq \Delta$ and $\text{ADOM}(abox_2(s_{02})) \subseteq \Delta$. A *context-sensitive jumping bisimulation* (CJ-Bisimulation) between Υ_1 and Υ_2 is a relation $\mathcal{B} \subseteq \Sigma_1 \times \Sigma_2$ such that $\langle s_1, s_2 \rangle \in \mathcal{B}$ implies that:

1. $s_1 =_{cx} s_2$, i.e., s_1 and s_2 are contextually equal (see Definition 6.44),
2. for each s'_1 , if $s_1 \Rightarrow_1 s'_1$ then there exist s'_2, t_1, \dots, t_n (for $n \geq 0$) with

$$s_2 \Rightarrow_2 t_1 \Rightarrow_2 \dots \Rightarrow_2 t_n \Rightarrow_2 s'_2$$

such that $\langle s'_1, s'_2 \rangle \in \mathcal{B}$, $\text{State}(temp) \notin abox_2(s'_2)$ and $\text{State}(temp) \in abox_2(t_i)$ for $i \in \{1, \dots, n\}$,

3. for each s'_2 , if

$$s_2 \Rightarrow_2 t_1 \Rightarrow_2 \dots \Rightarrow_2 t_n \Rightarrow_2 s'_2$$

(for $n \geq 0$) with $\text{State}(temp) \in abox_2(t_i)$ for $i \in \{1, \dots, n\}$ and $\text{State}(temp) \notin abox_2(s'_2)$, then there exists s'_1 with $s_1 \Rightarrow_1 s'_1$, such that $\langle s'_1, s'_2 \rangle \in \mathcal{B}$.

■

Let $\Upsilon_1 = \langle \Delta, T_{cx}, \Sigma_1, s_{01}, abox_1, ctx, \Rightarrow_1 \rangle$ be a context-sensitive transition system, and $\Upsilon_2 = \langle \Delta, T, \Sigma_2, s_{02}, abox_2, \Rightarrow_2 \rangle$ be a KB transition system, a state $s_1 \in \Sigma_1$ is *CJ-bisimilar* to $s_2 \in \Sigma_2$, written $s_1 \sim_{\text{CJ}} s_2$, if there exists a CJ-bisimulation relation \mathcal{B}

between \mathcal{T}_1 and \mathcal{T}_2 such that $\langle s_1, s_2 \rangle \in \mathcal{B}$. A transition system \mathcal{T}_1 is *CJ-bisimilar* to \mathcal{T}_2 , written $\mathcal{T}_1 \sim_{\text{CJ}} \mathcal{T}_2$, if there exists a CJ-bisimulation relation \mathcal{B} between \mathcal{T}_1 and \mathcal{T}_2 such that $\langle s_{01}, s_{02} \rangle \in \mathcal{B}$.

In the following lemmas we show some important properties of CJ-bisimilar states and transition systems that will be useful later to show that we can recast the verification of B-CSGKABs into S-GKABs.

Lemma 7.19. *Let $\mathcal{T}_1 = \langle \Delta, T_{cx}, \Sigma_1, s_{01}, abox_1, ctx, \Rightarrow_1 \rangle$ be a context-sensitive transition system, and $\mathcal{T}_2 = \langle \Delta, T, \Sigma_2, s_{02}, abox_2, \Rightarrow_2 \rangle$ be a KB transition system. Consider two states $s_1 \in \Sigma_1$ and $s_2 \in \Sigma_2$ such that $s_1 \sim_{\text{CJ}} s_2$. Then for every formula Φ of $\mu\mathcal{L}_{\text{CTX}}$, and every valuations v_1 and v_2 that assign to each of its free variables a constant $c_1 \in \text{ADOM}(abox_1(s_1))$ and $c_2 \in \text{ADOM}(abox_2(s_2))$, such that $c_1 = c_2$, we have that*

$$\mathcal{T}_1, s_1 \models \Phi_{v_1} \text{ if and only if } \mathcal{T}_2, s_2 \models t_j^{cx}(\Phi)v_2.$$

Proof. Similar to the combination of the proof of Lemmas 4.32 and 6.46. \square

Lemma 7.20. *Consider a context-sensitive transition system \mathcal{T}_1 , and a KB transition system \mathcal{T}_2 such that $\mathcal{T}_1 \sim_{\text{CJ}} \mathcal{T}_2$. For every closed $\mu\mathcal{L}_{\text{CTX}}$ formula Φ , we have:*

$$\mathcal{T}_1 \models \Phi \text{ if and only if } \mathcal{T}_2 \models t_j^{cx}(\Phi)$$

Proof. Let $\mathcal{T}_1 = \langle \Delta, T_{cx}, \Sigma_1, s_{01}, abox_1, ctx, \Rightarrow_1 \rangle$, and $\mathcal{T}_2 = \langle \Delta, T, \Sigma_2, s_{02}, abox_2, \Rightarrow_2 \rangle$. By the definition of CJ-bisimilar transition system we have that $s_{01} \sim_{\text{CJ}} s_{02}$. Thus, we obtain the proof as a consequence of Lemma 7.19, due to the fact that

$$\mathcal{T}_1, s_{01} \models \Phi \text{ if and only if } \mathcal{T}_2, s_{02} \models t_j^{cx}(\Phi)$$

\square

7.2.1.4 Reducing the Verification of B-CSGKABs into S-GKABs

In order to show that we can recast the verification of B-CSGKABs into S-GKABs, in the following two lemmas we show that the transition system of a B-CSGKABs \mathcal{G}_{cx} is CJ-bisimilar with the transition system of the corresponding S-GKAB $\tau_B^{cx}(\mathcal{G}_{cx})$ that is obtained via translation τ_B^{cx} .

Lemma 7.21. *Let \mathcal{G}_{cx} be a B-CSGKAB with transition system $\Upsilon_{\mathcal{G}_{cx}}^{f_{B}^{cx}}$, and let $\tau_B^{cx}(\mathcal{G}_{cx})$ be its corresponding S-GKAB (with transition system $\Upsilon_{\tau_B^{cx}(\mathcal{G}_{cx})}^{f_S}$) obtained through τ_B^{cx} . Consider a state $s_{cx} = \langle A_{cx}, m_{cx}, C, \delta_{cx} \rangle$ of $\Upsilon_{\mathcal{G}_{cx}}^{f_{B}^{cx}}$ and a state $s_s = \langle A_s, m_s, \delta_s \rangle$ of $\Upsilon_{\tau_B^{cx}(\mathcal{G}_{cx})}^{f_S}$. If $s_{cx} =_{cx} s_s$, $m_{cx} = m_s$ and $\delta_s = \kappa_B^{cx}(\delta_{cx})$, then $\langle A_{cx}, m_{cx}, C, \delta_{cx} \rangle \sim_{\text{CJ}} \langle A_s, m_s, \delta_s \rangle$.*

Proof. The proof is similar to the combination of the proof for Lemmas 5.33 and 6.48, by also considering the following:

1. The B-CSGKABs do the b-repair over the updated ABox and under the new context.
2. The context-sensitive b-repair program is executed after the ABox has been changed and the context has been updated.

3. By Theorem 7.17, we have that the result of the context-sensitive b-repair program is the same as the result of the b-repair computation. \square

Lemma 7.22. *Given a B-CSGKAB \mathcal{G}_{cx} , we have $\Upsilon_{\mathcal{G}_{cx}}^{f_{B}^{cx}} \sim_{\text{CJ}} \Upsilon_{\tau_B^{cx}(\mathcal{G}_{cx})}^{f_S}$*

Proof. Let

1. $\mathcal{G}_{cx} = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$, and
 $\Upsilon_{\mathcal{G}_{cx}}^{f_{B}^{cx}} = \langle \Delta, T_{cx}, \Sigma_{cx}, s_{0cx}, abox_{cx}, ctx, \Rightarrow_{cx} \rangle$,
2. $\tau_B^{cx}(\mathcal{G}_{cx}) = \langle T', A'_0, \Gamma', \delta' \rangle$ and $\Upsilon_{\tau_B^{cx}(\mathcal{G}_{cx})}^{f_S} = \langle \Delta, T', \Sigma_s, s_{0s}, abox_s, \Rightarrow_s \rangle$.

We have that $s_{0cx} = \langle A_0, m_{cx}, C_0, \delta \rangle$ and $s_{0s} = \langle A'_0, m_s, \delta' \rangle$ where $m_{cx} = m_s = \emptyset$. By the definition of κ_B^{cx} and τ_B^{cx} , we also have $s_{0cx} =_{cx} s_{0s}$, and $\delta' = \kappa_B^{cx}(\delta)$. Hence, by Lemma 7.21, we have $s_{0cx} \sim_{\text{CJ}} s_{0s}$. Therefore, by the definition of CJ-bisimulation, we have $\Upsilon_{\mathcal{G}_{cx}}^{f_{B}^{cx}} \sim_{\text{CJ}} \Upsilon_{\tau_B^{cx}(\mathcal{G}_{cx})}^{f_S}$. \square

Next, we show that the verification of $\mu\mathcal{L}_{\text{CTX}}$ properties over B-CSGKABs can be recast as verification of $\mu\mathcal{L}_A^{\text{EQL}}$ over S-GKABs as follows.

Theorem 7.23. *Given an B-CSGKAB \mathcal{G}_{cx} and a closed $\mu\mathcal{L}_{\text{CTX}}$ property Φ , we have*

$$\Upsilon_{\mathcal{G}_{cx}}^{f_{B}^{cx}} \models \Phi \text{ if and only if } \Upsilon_{\tau_B^{cx}(\mathcal{G}_{cx})}^{f_S} \models t_j^{cx}(\Phi)$$

Proof. By Lemma 7.22, we have that $\Upsilon_{\mathcal{G}_{cx}}^{f_{B}^{cx}} \sim_{\text{CJ}} \Upsilon_{\tau_B^{cx}(\mathcal{G}_{cx})}^{f_S}$. Hence, by Lemma 7.20, we have that for every $\mu\mathcal{L}_{\text{CTX}}$ property Φ

$$\Upsilon_{\mathcal{G}_{cx}}^{f_{B}^{cx}} \models \Phi \text{ if and only if } \Upsilon_{\tau_B^{cx}(\mathcal{G}_{cx})}^{f_S} \models t_j^{cx}(\Phi)$$

\square

7.2.2 From C-CSGKABs into Standard GKABs

Similar to the case of B-CSGKABs, a single transition in C-CSGKABs essentially update the ABox and context, and then apply c-repair to the newly updated ABox. Thus, to mimic the evolution of C-CSGKABs inside S-GKABs, we first adopt our approach in Section 6.5.2.1 to simulate the ABox and context evolution of C-CSGKABs within S-GKABs. Then, we adopt our approach in Section 5.3.2 to simulate the c-repair computation. As in the case of transforming B-CSGKABs into S-GKABs, we can not re-use the notion of c-repair action in Definition 5.36 directly because the TBox is evolving based on the context. Thus, we need to make the c-repair action able to adapt its behavior based on the context. I.e., we require that the c-repair action do the repair based on the TBox assertions that “hold” under the current context. To deal with this, we introduce a so called context-sensitive c-repair action that only consider those TBox assertions that “hold” under the current context and removes all ABox assertions that are involved in some form of inconsistency.

7.2.2.1 Translating C-CSGKABs into S-GKABs

As the first step towards translating C-CSGKABs into S-GKABs, in the following we introduce the notion of context-sensitive c-repair action.

Definition 7.24 (Context-sensitive C-Repair Action). Given a Contextualized TBox T_{cx} , let $\text{CTX}(\mathbb{ID})$ be the set of all possible contexts (see Definition 6.28). We define a 0-ary (i.e., has no action parameters) *context-sensitive c-repair action* $\alpha_c^{T_{cx}}$ over T_{cx} , where $\text{EFF}(\alpha_c^{T_{cx}})$ is the smallest set containing the following effects: For each context $C \in \text{CTX}(\mathbb{ID})$, we have:

*Context-sensitive
C-Repair Action*

- for each functionality assertion $(\text{funct } R) \in T_{cx}^C$, we have

$$q_C \wedge q_{\text{unsat}}^f((\text{funct } R), x, y, z) \rightsquigarrow \mathbf{del} \{R(x, y), R(x, z)\} \in \text{EFF}(\alpha_c^{T_{cx}})$$

- for each negative concept inclusion assertion $B_1 \sqsubseteq \neg B_2$ such that $T_{cx}^C \models B_1 \sqsubseteq \neg B_2$, we have

$$q_C \wedge q_{\text{unsat}}^n(B_1 \sqsubseteq \neg B_2, x) \rightsquigarrow \mathbf{del} \{B_1(x), B_2(x)\} \in \text{EFF}(\alpha_c^{T_{cx}});$$

- for each negative role inclusion assertion $R_1 \sqsubseteq \neg R_2$ such that $T_{cx}^C \models R_1 \sqsubseteq \neg R_2$, we have

$$q_C \wedge q_{\text{unsat}}^n(R_1 \sqsubseteq \neg R_2, x, y) \rightsquigarrow \mathbf{del} \{R_1(x, y), R_2(x, y)\} \in \text{EFF}(\alpha_c^{T_{cx}}).$$

- $\text{true} \rightsquigarrow \mathbf{del} \{\text{State}(\text{temp})\} \in \text{EFF}(\alpha_c^{T_{cx}}).$

■

As the last preliminary before we formally define the translation of C-CSGKABs into S-GKABs, in the following we define the program translation that will be used to translate the program in C-CSGKABs. Essentially we define a translation function κ_C^{cx} that concatenates each action invocation with a program that non-deterministically choose an action that changes the context, and then concatenates them with the context-sensitive c-repair action. Additionally, the translation function κ_C^{cx} also serves as a one-to-one correspondence (bijection) between the original and the translated program (as well as between the sub-program).

Definition 7.25 (Program Translation κ_C^{cx}). Given a C-CSGKABs $\mathcal{G}_{cx} = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$ we define a *translation* κ_C^{cx} that translates a program δ into a program δ' inductively as follows:

*Program Translation
 κ_C^{cx}*

$$\begin{aligned} \kappa_C^{cx}(\mathbf{pick} \langle Q(\vec{p}), \varphi_C \rangle . \alpha(\vec{p})) &= \mathbf{pick} Q'(\vec{p}) . \alpha'(\vec{p}); \delta_{\Pi_C}; \mathbf{pick} \text{true} . \alpha_c^{T_{cx}}() \\ \kappa_C^{cx}(\varepsilon) &= \varepsilon \\ \kappa_C^{cx}(\delta_1 | \delta_2) &= \kappa_C^{cx}(\delta_1) | \kappa_C^{cx}(\delta_2) \\ \kappa_C^{cx}(\delta_1; \delta_2) &= \kappa_C^{cx}(\delta_1); \kappa_C^{cx}(\delta_2) \\ \kappa_C^{cx}(\mathbf{if} \varphi \mathbf{then} \delta_1 \mathbf{else} \delta_2) &= \mathbf{if} \varphi \mathbf{then} \kappa_C^{cx}(\delta_1) \mathbf{else} \kappa_C^{cx}(\delta_2) \\ \kappa_C^{cx}(\mathbf{while} \varphi \mathbf{do} \delta) &= \mathbf{while} \varphi \mathbf{do} \kappa_C^{cx}(\delta) \end{aligned}$$

where

- $\mathbf{pick} Q'(\vec{p}) . \alpha'(\vec{p})$ is an action invocation obtained from $\mathbf{pick} \langle Q(\vec{p}), \varphi_C \rangle . \alpha(\vec{p})$ (see Definition 6.37),

- δ_{Π_C} is a context-change program obtained from Π_C as in Definition 6.39,
- $\alpha_c^{T_{cx}}$ is a context-sensitive c-repair action over T_{cx} as in Definition 7.24.

■

Having all of the machinery in hand, we are ready to define a translation τ_C^{cx} that, given a C-CSGKAB, produces an S-GKAB as follows:

Translation from
C-CSGKAB to
S-GKAB

Definition 7.26 (Translation from C-CSGKAB to S-GKAB). We define a translation τ_C^{cx} that, given a C-CSGKAB $\mathcal{G}_{cx} = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$, produces an S-GKAB $\tau_C^{cx}(\mathcal{G}_{cx}) = \langle T_{\mathbb{D}}, A_0 \cup A_{C_0}, \Gamma', \delta' \rangle$, where

- $T_{\mathbb{D}}$ is a TBox obtained from a set of context dimensions \mathbb{D} (see Definition 6.29),
- A_{C_0} is an ABox obtained from C_0 (see Definition 6.31),
- $\Gamma' = \Gamma_\alpha \cup \Gamma_C \cup \{\alpha_c^{T_{cx}}\}$ where:
 - Γ_α is obtained from Γ such that for each action $\alpha \in \Gamma$, we have $\alpha' \in \Gamma_\alpha$ where α' is a delayed action obtained from α (see Definition 6.36),
 - Γ_C is obtained from Π_C such that for each context-evolution rule $\langle Q, \varphi_C \rangle \mapsto C_{new}$ in Π_C , we have $\alpha_C \in \Gamma_C$ where α_C is an action obtained from the context-evolution rule $\langle Q, \varphi_C \rangle \mapsto C_{new}$ (see Definition 6.38),
 - $\alpha_c^{T_{cx}}$ is a context-sensitive c-repair action over T_{cx} (see Definition 7.24)
- $\delta' = \kappa_C^{cx}(\delta)$.

■

The $\mu\mathcal{L}_{\text{CTX}}$ property Φ over C-CSGKABs \mathcal{G}_{cx} can then be recast as a corresponding property over S-GKABs $\tau_C^{cx}(\mathcal{G}_{cx})$ using the following formula translation t_{trip} (see Definition 6.43). Utilizing those two translations, later we show that $\Upsilon_{\mathcal{G}_{cx}}^{f_{cx}^{cx}} \models \Phi$ if and only if $\Upsilon_{\tau_C^{cx}(\mathcal{G}_{cx})}^{f_{\tau_C^{cx}(\mathcal{G}_{cx})}^{cx}} \models t_{trip}(\Phi)$. As a consequence, we have that the verification of $\mu\mathcal{L}_{\text{CTX}}$ over C-CSGKABs can be reduced to the corresponding verification of $\mu\mathcal{L}_A^{\text{EQL}}$ over S-GKABs.

7.2.2.2 Reducing the Verification of C-CSGKABs into S-GKABs

In this subsection, we show that we can reduce the verification of C-CSGKABs into S-GKABs. To this aim, in the following we first show that the transition system of a C-CSGKABs \mathcal{G}_{cx} is ST-bisimilar with the transition system of the corresponding S-GKAB $\tau_C^{cx}(\mathcal{G}_{cx})$ that is obtained via translation τ_C^{cx} .

Lemma 7.27. *Let \mathcal{G}_{cx} be a C-CSGKAB with transition system $\Upsilon_{\mathcal{G}_{cx}}^{f_{\mathcal{G}_{cx}}^{cx}}$, and let $\tau_C^{cx}(\mathcal{G}_{cx})$ be its corresponding S-GKAB (with transition system $\Upsilon_{\tau_C^{cx}(\mathcal{G}_{cx})}^{f_{\tau_C^{cx}(\mathcal{G}_{cx})}^{cx}}$) obtained through τ_C^{cx} .*

Consider a state $s_{cx} = \langle A_{cx}, m_{cx}, C, \delta_{cx} \rangle$ of $\Upsilon_{\mathcal{G}_{cx}}^{f_{\mathcal{G}_{cx}}^{cx}}$ and a state $s_s = \langle A_s, m_s, \delta_s \rangle$ of $\Upsilon_{\tau_C^{cx}(\mathcal{G}_{cx})}^{f_{\tau_C^{cx}(\mathcal{G}_{cx})}^{cx}}$. If $s_{cx} =_{cx} s_s$, $m_{cx} = m_s$ and $\delta_s = \kappa_C^{cx}(\delta_{cx})$, then $\langle A_{cx}, m_{cx}, C, \delta_{cx} \rangle \sim_{\text{ST}} \langle A_s, m_s, \delta_s \rangle$.

Proof. The proof is similar to the combination of the proof for Lemmas 5.52 and 6.48, by also considering the following:

1. The C-CSGKABs do the c-repair over the updated ABox and under the new context.
2. The context-sensitive c-repair action is executed after the ABox has been changed and the context has been updated.

3. The different with the S-GKABs that capture S-CSGKABs is that after changing the context and materializing the ABox changes, instead of executing an action that check for inconsistency, the S-GKABs that capture C-CSGKABs execute the c-repair action which performs c-repair computation.
4. Similar to Theorem 5.51, we can also easily show the correctness of context-sensitive c-repair action. The important observation is that the context-sensitive c-repair action do the repair based on the context, i.e., it only consider those assertion in the TBox that “hold” under the corresponding context.

□

Lemma 7.28. *Given a C-CSGKAB \mathcal{G}_{cx} , we have $\Upsilon_{\mathcal{G}_{cx}}^{f_{\mathcal{C}}^{cx}} \sim_{\text{ST}} \Upsilon_{\tau_{\mathcal{C}}^{cx}(\mathcal{G}_{cx})}^{f_S}$*

Proof. Let

1. $\mathcal{G}_{cx} = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$, and

$$\Upsilon_{\mathcal{G}_{cx}}^{f_{\mathcal{C}}^{cx}} = \langle \Delta, T_{cx}, \Sigma_{cx}, s_{0cx}, abox_{cx}, ctx, \Rightarrow_{cx} \rangle,$$

2. $\tau_{\mathcal{C}}^{cx}(\mathcal{G}_{cx}) = \langle T', A'_0, \Gamma', \delta' \rangle$ and $\Upsilon_{\tau_{\mathcal{C}}^{cx}(\mathcal{G}_{cx})}^{f_S} = \langle \Delta, T', \Sigma_s, s_{0s}, abox_s, \Rightarrow_s \rangle$.

We have that $s_{0cx} = \langle A_0, m_c, C_0, \delta \rangle$ and $s_{0s} = \langle A'_0, m_s, \delta' \rangle$ where $m_c = m_s = \emptyset$. By the definition of $\kappa_{\mathcal{C}}^{cx}$ and $\tau_{\mathcal{C}}^{cx}$, we also have $s_{0cx} =_{cx} s_{0s}$, and $\delta' = \kappa_{\mathcal{C}}^{cx}(\delta)$. Hence, by Lemma 7.27, we have $s_{0cx} \sim_{\text{ST}} s_{0s}$. Therefore, by the definition of ST-bisimulation, we have $\Upsilon_{\mathcal{G}_{cx}}^{f_{\mathcal{C}}^{cx}} \sim_{\text{ST}} \Upsilon_{\tau_{\mathcal{C}}^{cx}(\mathcal{G}_{cx})}^{f_S}$. □

Having Lemma 7.28 in hand, in the following, we show that the verification of $\mu\mathcal{L}_{\text{CTX}}$ properties over C-CSGKABs can be recast as verification of $\mu\mathcal{L}_A^{\text{EQL}}$ properties over S-GKABs as follows.

Theorem 7.29. *Given a C-CSGKAB \mathcal{G}_{cx} and a closed $\mu\mathcal{L}_{\text{CTX}}$ property Φ , we have*

$$\Upsilon_{\mathcal{G}_{cx}}^{f_{\mathcal{C}}^{cx}} \models \Phi \text{ if and only if } \Upsilon_{\tau_{\mathcal{C}}^{cx}(\mathcal{G}_{cx})}^{f_S} \models t_{\text{trip}}(\Phi)$$

Proof. By Lemma 7.28, we have that $\Upsilon_{\mathcal{G}_{cx}}^{f_{\mathcal{C}}^{cx}} \sim_{\text{ST}} \Upsilon_{\tau_{\mathcal{C}}^{cx}(\mathcal{G}_{cx})}^{f_S}$. Hence, by Lemma 6.47, we have that the claim is proven. □

7.2.3 From E-CSGKABs into Standard GKABs

Similar to our reductions in Sections 7.2.1 and 7.2.2, to mimic the evolution of E-CSGKABs inside S-GKABs, we first adopt our approach in Section 6.5.2.1 to simulate the ABox and context evolution of E-CSGKABs within S-GKABs. Then we adopt our approach in Section 5.3.3 to simulate the computation of bold-evolution. In the following we highlight some important aspects on our transformation from E-CSGKABs into S-GKABs:

- We can not re-use the evolution action that we use to transform E-GKABs into S-GKABs in Section 5.3.3. The reason is simply because the TBox is changing based on the context. Thus, we require the evolution action to operate based on the current context that determines the TBox assertions that “hold” at a certain moment. Therefore, here we introduce the context-sensitive evolution action that simulate the bold-evolution computation while also aware of the context changing and adapts its computation based on the context.

- Similar to our reduction from E-GKABs into S-GKABs in Section 5.3.3, we need a mechanism to keep track the newly added assertions in order to perform bold-evolution. As mentioned earlier, to do that, we use those additional concept/role names that has been introduced to keep track the temporary information about ABox assertions to be added/deleted. In particular, we use the ABox assertions that are made by the added fact marker concept names (i.e., those one with superscript a). Additionally, those kind of ABox assertions are also useful for checking the consistency of the newly added assertions.
- As before, we also reserve a special concept assertion $\text{State}(\text{temp})$ in order to mark the intermediate states.

7.2.3.1 Translating E-CSGKABs into S-GKABs

This section is aimed to show how we transform E-CSGKABs into S-GKABs. To open this section, we start by introducing the notion of duplicated action and duplicated action invocation that is obtained from context-evolution rules. Basically they are similar to the one in Definition 6.38, except that for each newly added concept assertion $N(c)$, we do not delete the corresponding concept assertion $N^a(c)$ where by N^a is an added fact marker concept name (similarly for roles). The purpose of this modification are to keep the information about the newly added ABox assertions and to enable the possibility to check the consistency of the update.

Definition 7.30 (Duplicated Action and Duplicated Action Invocation Obtained From Context-evolution Rule). A *duplicated action invocation obtained from a context-evolution rule* $\langle Q, \varphi_C \rangle \mapsto C_{\text{new}}$ in Π_C , is an action invocation **pick** $Q'.\alpha_C^d()$ where

1. $Q' = Q_{cx} \wedge q_{\varphi_C}$ where Q_{cx} is contextually compiled query of Q , and q_{φ_C} is the query obtained from the context expression φ_C .
2. α_C^d is a 0-ary action obtained from $\langle Q, \varphi_C \rangle \mapsto C_{\text{new}}$ as follows:
 - (a) For each $[d_i \mapsto v_j] \in C_{\text{new}}$, we have:
 - i. $\text{true} \rightsquigarrow \mathbf{add} \{D_i^{v_j}(c), \mathbf{D}_i^{v_j}(c)\}$ in $\text{EFF}(\alpha_C^d)$, and
 - ii. $\text{true} \rightsquigarrow \mathbf{del} \{D_i^{v_k}(c), \mathbf{D}_i^{v_k}(c)\}$ in $\text{EFF}(\alpha_C^d)$ for every $v_k \in \text{Dom}(d_i)$ such that $v_k \neq v_j$.
 - (b) For each concept name $N \in \text{VOC}(T_{cx})$, we have
 - i. $N^a(x) \rightsquigarrow \mathbf{add} \{N(x)\}$ in $\text{EFF}(\alpha_C^d)$,
 - ii. $N^d(x) \rightsquigarrow \mathbf{del} \{N(x), N^d(x)\}$ in $\text{EFF}(\alpha_C^d)$.

Compare to Definition 6.38, the different is that in (i) for each newly added ABox assertion formed by concept N we do not delete the ABox assertion that is formed by N^a . As mentioned before, essentially, the assertion formed by N^a acts as a marker that marks the newly added assertion and we still need their information later.

- (c) Similarly for the role names, we create the same effect as in the step (b) above.

In this case we say that α_C^d is a *duplicated action obtained from the context-evolution rule* $\langle Q, \varphi_C \rangle \mapsto C_{\text{new}}$. ■

We now proceed to lift the notion of evolution action in Definition 5.55 into the context-sensitive evolution action as follows:

Duplicated Action
and Duplicated
Action Invocation
Obtained From
Context-evolution
Rule

Definition 7.31 (Context-sensitive Evolution Action). Given a Contextualized TBox T_{cx} , let $\text{CTX}(\mathbb{D})$ be the set of all possible context (see Definition 6.28). We define a 0-ary (i.e., has no action parameters) *context-sensitive evolution action* $\alpha_e^{T_{cx}}$ over T_{cx} , where $\text{EFF}(\alpha_e^{T_{cx}})$ is the smallest set containing the following effects: For each context $C \in \text{CTX}(\mathbb{D})$, we have:

*Context-sensitive
Evolution Action*

- for each assertion $(\text{funct } R) \in T_{cx}^C$, we have

$$q_C \wedge \exists z. q_{\text{unsat}}^f((\text{funct } R), x, y, z) \wedge R^a(x, y) \rightsquigarrow \mathbf{del} \{R(x, z)\} \in \text{EFF}(\alpha_e^{T_{cx}}),$$

- for each negative concept inclusion assertion $B_1 \sqsubseteq \neg B_2$ such that $T_{cx}^C \models B_1 \sqsubseteq \neg B_2$, we have

$$q_C \wedge q_{\text{unsat}}^n(B_1 \sqsubseteq \neg B_2, x) \wedge B_1^a(x) \rightsquigarrow \mathbf{del} \{B_2(x)\} \in \text{EFF}(\alpha_e^{T_{cx}}),$$

- for each negative role inclusion assertion $R_1 \sqsubseteq \neg R_2$ such that $T_{cx}^C \models R_1 \sqsubseteq \neg R_2$, we have:

$$q_C \wedge q_{\text{unsat}}^n(R_1 \sqsubseteq \neg R_2, x, y) \wedge R_1^a(x, y) \rightsquigarrow \mathbf{del} \{R_2(x, y)\} \in \text{EFF}(\alpha_e^{T_{cx}}),$$

- for each concept name $N \in \text{VOC}(T_{cx})$, we have:

$$N^a(x) \rightsquigarrow \mathbf{del} \{N^a(x)\} \in \text{EFF}(\alpha_e^{T_{cx}}),$$

where N^a is the reserved added fact marker concept name for N .

- for each role name $P \in \text{VOC}(T_{cx})$, we have:

$$P^a(x, y) \rightsquigarrow \mathbf{del} \{P^a(x, y)\} \in \text{EFF}(\alpha_e^{T_{cx}}),$$

where P^a is the reserved added fact marker concept name for P .

- $\text{true} \rightsquigarrow \mathbf{del} \{\text{State}(\text{temp})\} \in \text{EFF}(\alpha_e^{T_{cx}})$.

■

As the last preliminary before we formally define the translation of E-CSGKABs into S-GKABs, in the following we define the program translation that will be used to translate the program in E-CSGKABs. Essentially we define a translation function κ_E^{cx} that concatenates each action invocation with a program that non-deterministically choose an action that changes the context, and then concatenates them with the update consistency checker action, and also with the context-sensitive evolution action. Additionally, the translation function κ_E^{cx} also serves as a one-to-one correspondence (bijection) between the original and the translated program (as well as between the sub-program).

Definition 7.32 (Program Translation κ_E^{cx}). Given an E-CSGKABs $\mathcal{G}_{cx} = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$ we define a *translation* κ_E^{cx} that translates a program δ into a program δ' inductively as follows:

*Program Translation
 κ_E^{cx}*

$$\begin{aligned} \kappa_E^{cx}(\mathbf{pick} \langle Q(\vec{p}), \varphi_C \rangle. \alpha(\vec{p})) &= \mathbf{pick} \ Q'(\vec{p}). \alpha'(\vec{p}); \delta_{\Pi_C}; \mathbf{pick} \ \neg Q_{\text{unsat}}^{T_a} \text{ECQ}. \alpha_e^{T_{cx}}() \\ \kappa_E^{cx}(\varepsilon) &= \varepsilon \\ \kappa_E^{cx}(\delta_1 | \delta_2) &= \kappa_E^{cx}(\delta_1) | \kappa_E^{cx}(\delta_2) \\ \kappa_E^{cx}(\delta_1; \delta_2) &= \kappa_E^{cx}(\delta_1); \kappa_E^{cx}(\delta_2) \\ \kappa_E^{cx}(\mathbf{if} \ \varphi \ \mathbf{then} \ \delta_1 \ \mathbf{else} \ \delta_2) &= \mathbf{if} \ \varphi \ \mathbf{then} \ \kappa_E^{cx}(\delta_1) \ \mathbf{else} \ \kappa_E^{cx}(\delta_2) \\ \kappa_E^{cx}(\mathbf{while} \ \varphi \ \mathbf{do} \ \delta) &= \mathbf{while} \ \varphi \ \mathbf{do} \ \kappa_E^{cx}(\delta) \end{aligned}$$

where

- **pick** $Q'(\vec{p}).\alpha'(\vec{p})$ is a action invocation obtained from **pick** $\langle Q(\vec{p}), \varphi_C \rangle.\alpha(\vec{p})$ as in Definition 6.37.
- δ_{Π_C} is a context-change program obtained from Π_C as in Definition 6.39, except that it is formed by duplicated action invocation obtained from context evolution rule as in Definition 7.30.
- $\alpha_e^{T_{cx}}$ is a context-sensitive evolution action over T_{cx} as in Definition 7.31.
- $Q_{\text{unsatECQ}}^{T_a}$ is a context-sensitive Q-UNSAT-ECQ over T_a (see Definition 7.11), where T_a is obtained from T_{cx} by renaming each concept name N in T_{cx} into N^a (similarly for roles). Thus, with this mechanism, we can block any further execution when the newly added assertions are inconsistent. ■

Having all of the machinery in hand, we are ready to define a translation τ_E^{cx} that, given a E-CSGKAB, produces an S-GKAB as follows:

Translation from
E-CSGKAB to
S-GKAB

Definition 7.33 (Translation from E-CSGKAB to S-GKAB). We define a translation τ_E^{cx} that, given an E-CSGKAB $\mathcal{G}_{cx} = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$, produces an S-GKAB $\tau_E^{cx}(\mathcal{G}_{cx}) = \langle T_D, A_0 \cup A_{C_0}, \Gamma', \delta' \rangle$, where

- T_D is a TBox obtained from a set of context dimensions \mathbb{D} (see Definition 6.29),
- A_{C_0} is an ABox obtained from C_0 (see Definition 6.31),
- $\Gamma' = \Gamma_\alpha \cup \Gamma_C \cup \{\alpha_e^{T_{cx}}\}$ where:
 - Γ_α is obtained from Γ such that for each action $\alpha \in \Gamma$, we have $\alpha' \in \Gamma_\alpha$ where α' is a delayed action obtained from α (see Definition 6.36),
 - Γ_C is obtained from Π_C such that for each context-evolution rule $\langle Q, \varphi_C \rangle \mapsto C_{new}$ in Π_C , we have $\alpha_C \in \Gamma_C$ where α_C is a *duplicated action obtained from the context-evolution rule* $\langle Q, \varphi_C \rangle \mapsto C_{new}$ (see Definition 7.30),
 - $\alpha_e^{T_{cx}}$ is a context-sensitive evolution action over T_{cx} (see Definition 7.31)
- $\delta' = \kappa_E^{cx}(\delta)$. ■

A $\mu\mathcal{L}_{\text{CTX}}$ property Φ over E-CSGKABs \mathcal{G}_{cx} can then be recast as a corresponding property over $\tau_E^{cx}(\mathcal{G}_{cx})$ by using the formula translation t_{trip} (see Definition 6.43). Employing those two translations, later we show that $\Upsilon_{\mathcal{G}_{cx}}^{f_{E}^{cx}} \models \Phi$ if and only if $\Upsilon_{\tau_E^{cx}(\mathcal{G}_{cx})}^{f_S} \models t_{trip}(\Phi)$. As a consequence, we have that the verification of $\mu\mathcal{L}_{\text{CTX}}$ over E-CSGKABs can be reduced to the corresponding verification of $\mu\mathcal{L}_A^{\text{EQL}}$ over S-GKABs.

7.2.3.2 Reducing the Verification of E-CSGKABs into S-GKABs

We now advanced further to show that the verification of E-CSGKABs can be recast into the verification of S-GKABs. Below, we first show that the transition system of a E-CSGKABs \mathcal{G}_{cx} is ST-bisimilar with the transition system of the corresponding S-GKAB $\tau_E^{cx}(\mathcal{G}_{cx})$ that is obtained via translation τ_E^{cx} .

Lemma 7.34. *Let \mathcal{G}_{cx} be an E-CSGKAB with transition system $\Upsilon_{\mathcal{G}_{cx}}^{f_{E}^{cx}}$, and let $\tau_E^{cx}(\mathcal{G}_{cx})$ be its corresponding S-GKAB (with transition system $\Upsilon_{\tau_E^{cx}(\mathcal{G}_{cx})}^{f_S}$) obtained through τ_E^{cx} . Consider a state $s_{cx} = \langle A_{cx}, m_{cx}, C, \delta_{cx} \rangle$ of $\Upsilon_{\mathcal{G}_{cx}}^{f_{E}^{cx}}$ and a state $s_s = \langle A_s, m_s, \delta_s \rangle$ of*

$\Upsilon_{\tau_E^{cx}(\mathcal{G}_{cx})}^{fs}$. If $s_{cx} =_{cx} s_s$, $m_{cx} = m_s$ and $\delta_s = \kappa_E^{cx}(\delta_{cx})$, then $\langle A_{cx}, m_{cx}, C, \delta_{cx} \rangle \sim_{ST} \langle A_s, m_s, \delta_s \rangle$.

Proof. The proof is similar to the combination of the proof for Lemmas 5.60 and 6.48, by also considering the following:

1. The different with the S-GKABs that capture S-CSGKABs is that after changing the context and materializing the ABox changes, instead of executing an action that checks the inconsistency, the S-GKABs that capture E-CSGKABs execute the evolution action which performs bold-evolution computation.
2. The context-sensitive evolution action is executed after the the context has been updated. This is aligned with Definition 7.7 that E-CSGKABs perform the bold evolution w.r.t. the TBox under the new context. Additionally, as it can be seen from the translation κ_E^{cx} (see Definition 7.32), before executing the evolution action, we also check the consistency of the updates w.r.t. the TBox under the new context. This guarantees that we fulfill the requirement in Definition 7.7 that the updates must be consistent w.r.t. the TBox under the new context.
3. Similar to Lemmas 5.58 and 5.59, we can also easily show the correctness of context-sensitive evolution action that it performs the bold-evolution computation. The important observation is that the context-sensitive evolution action performs the bold-evolution based on the context, i.e., it only consider those assertions in the TBox that “hold” under the corresponding context.

□

Lemma 7.35. *Given an E-CSGKAB \mathcal{G}_{cx} , we have $\Upsilon_{\mathcal{G}_{cx}}^{f_{E}^{cx}} \sim_{ST} \Upsilon_{\tau_E^{cx}(\mathcal{G}_{cx})}^{fs}$*

Proof. Let

1. $\mathcal{G}_{cx} = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$, and $\Upsilon_{\mathcal{G}_{cx}}^{f_{E}^{cx}} = \langle \Delta, T_{cx}, \Sigma_{cx}, s_{0cx}, abox_{cx}, ctx, \Rightarrow_{cx} \rangle$,
2. $\tau_E^{cx}(\mathcal{G}_{cx}) = \langle T', A'_0, \Gamma', \delta' \rangle$ and $\Upsilon_{\tau_E^{cx}(\mathcal{G}_{cx})}^{fs} = \langle \Delta, T', \Sigma_s, s_{0s}, abox_s, \Rightarrow_s \rangle$.

We have that $s_{0cx} = \langle A_0, m_c, C_0, \delta \rangle$ and $s_{0s} = \langle A'_0, m_s, \delta' \rangle$ where $m_c = m_s = \emptyset$. By the definition of κ_E^{cx} and τ_E^{cx} , we also have $s_{0cx} =_{cx} s_{0s}$, and $\delta' = \kappa_E^{cx}(\delta)$. Hence, by Lemma 7.27, we have $s_{0cx} \sim_{ST} s_{0s}$. Therefore, by the definition of ST-bisimulation, we have $\Upsilon_{\mathcal{G}_{cx}}^{f_{E}^{cx}} \sim_{ST} \Upsilon_{\tau_E^{cx}(\mathcal{G}_{cx})}^{fs}$. □

We now proceed to show that the verification of $\mu\mathcal{L}_{CTX}$ properties over E-CSGKABs can be recast as verification of $\mu\mathcal{L}_A^{EQL}$ over S-GKABs as follows.

Theorem 7.36. *Given an E-CSGKAB \mathcal{G}_{cx} and a closed $\mu\mathcal{L}_{CTX}$ property Φ , we have*

$$\Upsilon_{\mathcal{G}_{cx}}^{f_{E}^{cx}} \models \Phi \text{ if and only if } \Upsilon_{\tau_E^{cx}(\mathcal{G}_{cx})}^{fs} \models t_{trip}(\Phi)$$

Proof. By Lemma 7.35, we have that $\Upsilon_{\mathcal{G}_{cx}}^{f_{E}^{cx}} \sim_{ST} \Upsilon_{\tau_E^{cx}(\mathcal{G}_{cx})}^{fs}$. Hence, by Lemma 6.47, we have that the claim is proven. □

7.2.4 Bring It All Together: Verification of I-CSGKABs

To sum up, we state the result of I-CSGKABs verification as follows:

Theorem 7.37. *Verification of $\mu\mathcal{L}_{\text{CTX}}$ properties over I-CSGKABs can be recast as verification over S-GKABs.*

Proof. As a consequence of Theorems 7.23, 7.29 and 7.36, we essentially show that the verification of $\mu\mathcal{L}_{\text{CTX}}$ properties over I-CSGKABs can be recast as verification over S-GKABs since we can recast the verification of $\mu\mathcal{L}_{\text{CTX}}$ properties over B-CSGKABs, C-CSGKABs, and E-CSGKABs as verification over S-GKABs. \square

From Theorems 4.54 and 7.37, we get our next result that verification of inconsistency-aware variants of CSGKABs introduced in Section 7.1 can be compiled into verification of KABs, by first reducing verification of I-CSGKABs into verification of S-GKABs, and then reducing verification of S-GKABs into verification of KABs.

Theorem 7.38. *Verification of $\mu\mathcal{L}_{\text{CTX}}$ properties over I-CSGKABs can be recast as verification over KABs.*

Proof. The proof is easily obtained from the Theorems 4.54 and 7.37, since by Theorem 7.37 we can recast the verification of $\mu\mathcal{L}_{\text{CTX}}$ over I-CSGKABs as verification over S-GKABs and then by Theorem 4.54 we can recast the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ over S-GKABs as verification over KABs. Thus, combining those two ingredients, we can reduce the verification of $\mu\mathcal{L}_{\text{CTX}}$ over I-CSGKABs into the corresponding verification of $\mu\mathcal{L}_A^{\text{EQL}}$ over KABs. \square

7.2.5 Verification of Run-bounded I-CSGKABs

We now aim to show that the reductions from I-CSGKABs to S-GKABs preserve run-boundedness.

Lemma 7.39. *Let \mathcal{G}_{cx} be a B-CSGKAB and $\tau_B^{cx}(\mathcal{G}_{cx})$ be its corresponding S-GKAB. We have \mathcal{G}_{cx} is run-bounded if and only if $\tau_B^{cx}(\mathcal{G}_{cx})$ is run-bounded.*

Proof. Let

1. $\mathcal{G}_{cx} = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$, and $\mathcal{T}_{\mathcal{G}_{cx}}^{f_{cx}}$ be its transition system,
2. $\mathcal{T}_{\tau_B^{cx}(\mathcal{G}_{cx})}^{f_S}$ the transition system of $\tau_B^{cx}(\mathcal{G}_{cx})$.

The proof is easily obtained since

- the translation τ_B^{cx} essentially only appends each action invocation in δ with some additional programs to handle the context change and manage inconsistency.
- the program that manage inconsistency never inject new additional constants, but only remove facts causing inconsistency,
- the program that is used to simulate the context evolution does not inject unbounded number of new constants. In fact, we only reserve a constant \mathbf{c} to simulate the context (i.e., to construct the ABox assertions that represent the context dimension assignments).

- by Lemma 7.22, we have that $\mathcal{R}_{\mathcal{G}_{cx}}^{f_{cx}^B} \sim_{\text{CJ}} \tau_B^{cx}(\mathcal{G}_{cx})$. Thus, basically they are “equivalent” modulo intermediate states (states containing $\text{State}(\text{temp})$) and also by considering that they represent context information in a different way (i.e., each two bisimilar states are equivalent modulo context ABox assertions).

□

Lemma 7.40. *Let \mathcal{G}_{cx} be a C-CSGKAB and $\tau_C^{cx}(\mathcal{G}_{cx})$ be its corresponding S-GKAB. We have \mathcal{G}_{cx} is run-bounded if and only if $\tau_C^{cx}(\mathcal{G}_{cx})$ is run-bounded.*

Proof. Similar to the proof of Lemma 7.39. □

Lemma 7.41. *Let \mathcal{G}_{cx} be an E-CSGKAB and $\tau_E^{cx}(\mathcal{G}_{cx})$ be its corresponding S-GKAB. We have \mathcal{G}_{cx} is run-bounded if and only if $\tau_E^{cx}(\mathcal{G}_{cx})$ is run-bounded.*

Proof. Similar to the proof of Lemma 7.39. □

Finally, we show the result on the verification of $\mu\mathcal{L}_{\text{CTX}}$ properties over run-bounded I-CSGKABs as follows.

Theorem 7.42. *Verification of $\mu\mathcal{L}_{\text{CTX}}$ properties over run-bounded I-CSGKABs is decidable, and reducible to standard μ -calculus finite-state model checking.*

Proof. By Lemmas 7.39 to 7.41, the translation from I-CSGKABs to S-GKABs preserves run-boundedness. Thus, the claim follows by combining Theorem 7.37 and Theorem 4.56. □

7.3 From Standard GKABs to Inconsistency-aware Context-sensitive GKABs

We have seen so far that we can transform I-CSGKABs into S-GKABs and recast the verification of I-CSGKABs into S-GKABs. Now, we show that we can also do the other direction. I.e., we show that we can recast the verification of S-GKABs into the verification of I-CSGKABs. As a consequence, we have that S-GKABs and I-CSGKABs are reducible to each other in terms of the verification.

The general strategy to compile S-GKABs into I-CSGKABs is as follows:

- Basically we combine the approach in Sections 5.4 and 6.6.
- We force the TBox to stay the same for all of the states along the system evolution by preventing the context change.
- We prevent the repair when we encounter an inconsistent state and force to reject each action execution that leads to an inconsistent state.

Here we only show how we can reduce the verification of S-GKABs into B-CSGKABs. The reductions from S-GKABs into C-CSGKABs and E-CSGKABs are similar.

To realize the strategy above, in the following we fix a set \mathbb{ID} of context dimension containing only a single context dimension d (i.e., $\mathbb{ID} = \{d\}$). Moreover, $d \in \mathbb{ID}$ has a tree shaped finite value domain $\langle \text{Dom}(d), \prec_d \rangle$ where $\text{Dom}(d)$ contains only a single value \top_d (i.e., $\text{Dom}(d) = \top_d$).

We now introduce the translation for program in S-GKABs. Particularly, we define a translation function κ_{sic} that basically

1. replaces each action invocation with a context-sensitive action invocation in which its context expression always holds in any context, and then
2. concatenates it with an action invocation that does the inconsistency check.

Additionally, the translation function κ_{sic} also serves as a one-to-one correspondence (bijection) between the original and the translated program (as well as between the sub-program).

Program Translation
 κ_{sic}

Definition 7.43 (Program Translation κ_{sic}). Given a set of actions Γ , a program δ over Γ , and a TBox T , we define a *translation* κ_{sic} which translates a program into a program inductively as follows:

$$\begin{aligned}
 \kappa_{sic}(\mathbf{pick} \ Q(\vec{p}).\alpha(\vec{p})) &= \mathbf{pick} \ \langle Q(\vec{p}), \varphi_C \rangle.\alpha'(\vec{p}); \mathbf{pick} \ \neg Q_{\text{unsatECQ}}^T.\alpha_{\perp}() \\
 \kappa_{sic}(\varepsilon) &= \varepsilon \\
 \kappa_{sic}(\delta_1 | \delta_2) &= \kappa_{sic}(\delta_1) | \kappa_{sic}(\delta_2) \\
 \kappa_{sic}(\delta_1; \delta_2) &= \kappa_{sic}(\delta_1); \kappa_{sic}(\delta_2) \\
 \kappa_{sic}(\mathbf{if} \ \varphi \ \mathbf{then} \ \delta_1 \ \mathbf{else} \ \delta_2) &= \mathbf{if} \ \varphi \ \mathbf{then} \ \kappa_{sic}(\delta_1) \ \mathbf{else} \ \kappa_{sic}(\delta_2) \\
 \kappa_{sic}(\mathbf{while} \ \varphi \ \mathbf{do} \ \delta) &= \mathbf{while} \ \varphi \ \mathbf{do} \ \kappa_{sic}(\delta)
 \end{aligned}$$

where

- Q_{unsatECQ}^T is a boolean Q-UNSAT-ECQ over T (similar to Q-UNSAT-FOL in Definition 2.43) that is used to check the inconsistency. It will be evaluated to true if the ABox is T -inconsistent.
- $\varphi_C = \{[d \rightsquigarrow \top_d]\}$
- action $\alpha'(\vec{p})$ is obtained from $\alpha(\vec{p}) \in \Gamma$, such that

$$\text{EFF}(\alpha') = \text{EFF}(\alpha) \cup \{\mathbf{true} \rightsquigarrow \mathbf{add} \ \{\text{State}(\text{temp})\}\}$$

- α_{\perp} is a 0-ary action of the form

$$\alpha_{\perp}() : \{\mathbf{true} \rightsquigarrow \mathbf{del} \ \{\text{State}(\text{temp})\}\}$$

■

We then define the following translation that transform S-GKABs into B-CSGKABs as follows.

Translation from
S-GKAB to
B-CSGKAB

Definition 7.44 (Translation from S-GKAB to B-CSGKAB). We define a translation τ_{sic} that, given an S-GKAB $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$, produces a B-CSGKAB $\tau_{sic}(\mathcal{G}) = \langle T_{cx}, A_0, \Gamma', \delta', C_0, \Pi_C \rangle$, where

- T_{cx} is obtained from T such that for each positive inclusion assertion $t \in T$, we have $\langle t : \varphi \rangle$ where $\varphi = [d \rightsquigarrow \top_d]$,
- $\Gamma' = \Gamma_{\alpha} \cup \{\alpha_{\perp}\}$ where
 - Γ_{α} is obtained from Γ such that for each $\alpha \in \Gamma$, we have $\alpha' \in \Gamma_{\alpha}$ where $\text{EFF}(\alpha') = \text{EFF}(\alpha) \cup \{\mathbf{true} \rightsquigarrow \mathbf{add} \ \{\text{State}(\text{temp})\}\}$,
 - α_{\perp} is a 0-ary action of the form $\alpha_{\perp}() : \{\mathbf{true} \rightsquigarrow \mathbf{del} \ \{\text{State}(\text{temp})\}\}$.
- $\delta' = \kappa_{sic}(\delta)$.

- $C_0 = \{[d \rightsquigarrow \top_d]\}$,
- $\Pi_C = \{\langle \text{true}, [d \rightsquigarrow \top_d] \rangle \mapsto \{[d \rightsquigarrow \top_d]\}\}$

■

Next, we show that given an S-GKAB \mathcal{G} , and $\mu\mathcal{L}_A^{\text{EQL}}$ formula Φ , we have that $\Upsilon_{\mathcal{G}}^{fs} \models \Phi$ if and only if $\Upsilon_{\tau_{sic}(\mathcal{G})}^{f_{B}^{cx}} \models \Phi$. The strategy is as follows:

1. Recall the notion of S-Bisimulation in Section 5.3.2.1. Here we use a similar notion of bisimulation except that now the bisimulation relation is defined between a KB transition system and a context-sensitive transition system. However, the bisimulation condition are kept the same. Therefore, for compactness of presentation, here we do not redefine a new bisimulation relation. All notions related to S-Bisimulation that was introduced in Section 5.3.2.1 can be seamlessly recast into this setting.
2. To utilize the S-Bisimulation relation and its properties, in the following we show that given an S-GKAB, its transition system is S-bisimilar to the transition system of its corresponding B-CSGKAB that is obtained through τ_{sic} . Then, by using Lemma 5.42 (except that now we consider a KB transition system and a context-sensitive transition system) and also by considering that $\mu\mathcal{L}_{\text{CTX}}$ without context expression is the same as $\mu\mathcal{L}_A^{\text{EQL}}$, we can easily recast the verification of S-GKABS into B-CSGKABS.

In the following two lemmas we intend to show that given an S-GKAB, its transition system is S-bisimilar to the transition system of its corresponding B-CSGKAB that is obtained through τ_{sic} .

Lemma 7.45. *Let \mathcal{G} be an S-GKAB with transition system $\Upsilon_{\mathcal{G}}^{fs}$, and let $\tau_{sic}(\mathcal{G})$ be its corresponding B-CSGKAB (with transition system $\Upsilon_{\tau_{sic}(\mathcal{G})}^{f_{B}^{cx}}$) obtained through τ_{sic} . Consider a state $s_s = \langle A_s, m_s, \delta_s \rangle$ of $\Upsilon_{\mathcal{G}}^{fs}$, and a state $s_{cx} = \langle A_{cx}, m_{cx}, C, \delta_{cx} \rangle$ of $\Upsilon_{\tau_{sic}(\mathcal{G})}^{f_{B}^{cx}}$. If $A_{cx} = A_s$, $m_{cx} = m_s$, and $\delta_{cx} = \kappa_{sic}(\delta_s)$, then $\langle A_{cx}, m_{cx}, C, \delta_{cx} \rangle \sim_{\text{so}} \langle A_s, m_s, \delta_s \rangle$.*

Proof. Now, let

1. $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ and $\Upsilon_{\mathcal{G}}^{fs} = \langle \Delta, T, \Sigma_s, s_{0s}, \text{abox}_s, \Rightarrow_s \rangle$.
2. $\tau_{sic}(\mathcal{G}) = \langle T_{cx}, A_0, \Gamma', \delta', C_0, \Pi_C \rangle$, and $\Upsilon_{\tau_{sic}(\mathcal{G})}^{f_{B}^{cx}} = \langle \Delta, T_{cx}, \Sigma_{cx}, s_{0cx}, \text{abox}_{cx}, \text{ctx}, \Rightarrow_{cx} \rangle$,

Now, we have to show the following: For every state $\langle A'_s, m'_s, \delta'_s \rangle$ such that $\langle A_s, m_s, \delta_s \rangle \Rightarrow_s \langle A'_s, m'_s, \delta'_s \rangle$ there exists $\langle A'_{cx}, m'_{cx}, C', \delta'_{cx} \rangle$ such that

- (a) we have $\langle A_{cx}, m_{cx}, C, \delta_{cx} \rangle \Rightarrow \langle A'_{cx}, m'_{cx}, C', \delta'_{cx} \rangle$,
- (b) $A'_s = A'_{cx}$
- (c) $m'_s = m'_{cx}$;
- (d) $\delta'_{cx} = \kappa_{sic}(\delta'_s)$.

The proof can be easily obtained by considering that within B-CSGKAB, by the definition of τ_{sic} (see Definition 7.44), it is easy to see that the following hold:

- The initial context is $C_0 = \{[d \rightsquigarrow \top_d]\}$ and basically this is the only one possible context in the system. Furthermore, the context stays the same along the system evolution because we only have a single context evolution rule $\langle \text{true}, [d \rightsquigarrow \top_d] \rangle \mapsto \{[d \rightsquigarrow \top_d]\}$ that never change the context. Hence, the TBox stay the same for all states. Furthermore, since for each $\langle t : \varphi \rangle \in T_{cx}$ we

have $\varphi = [d \rightsquigarrow \top_d]$, it is easy to see that all of the TBox assertions hold in our only one possible context. As a consequence, essentially the situation of the TBox is the same as in the original S-GKAB.

- We basically can ignore the context expression that guards each context-sensitive action invocation in δ' because it is always be satisfied in any case. Hence, each context-sensitive action invocation in δ' is essentially the same as the usual action invocation in δ .
- Each state in the transition system of B-CSGKAB is always consistent, because we only keep the positive inclusion assertions when we translate an S-GKAB into a B-CSGKAB. Thus, the repair mechanism in B-GKAB will not change anything.
- We transform each action invocation in the program in the given S-GKAB such that it will always be followed by the action invocation **pick** $\neg Q_{\text{unsatECQ}}^T \cdot \alpha_{\perp}()$ where Q_{unsatECQ}^T will be evaluated to true when the corresponding ABox is T -inconsistent. Hence, the inconsistency check is basically delegated to the evaluation of the query that acts as the guard of the action α_{\perp} and it is triggered after each action execution. The action α_{\perp} will not be executed if the previous action execution leads into an inconsistent state w.r.t. the TBox T . Thus, it is easy to see that when an action execution in S-GKAB is blocked because it leads into a T -inconsistent state, then the corresponding action execution in B-CSGKAB will not lead into a new state without $\text{State}(\text{temp})$ as well. However, when an execution in S-GKAB leads into a new T -consistent state, the corresponding action execution in B-CSGKAB will be followed by the execution of α_{\perp} and it leads into a new state without $\text{State}(\text{temp})$.

□

Lemma 7.46. *Given an S-GKAB \mathcal{G} , we have $\Upsilon_{\mathcal{G}}^{fs} \sim_{\text{so}} \Upsilon_{\tau_{\text{sic}}(\mathcal{G})}^{f_{\text{B}}^{cx}}$*

Proof. Let

1. $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$ and $\Upsilon_{\mathcal{G}}^{fs} = \langle \Delta, T, \Sigma_s, s_{0s}, \text{abox}_s, \Rightarrow_s \rangle$.
2. $\tau_{\text{sic}}(\mathcal{G}) = \langle T_{cx}, A_0, \Gamma', \delta', C_0, \Pi_C \rangle$, and $\Upsilon_{\tau_{\text{sic}}(\mathcal{G})}^{f_{\text{B}}^{cx}} = \langle \Delta, T_{cx}, \Sigma_{cx}, s_{0cx}, \text{abox}_{cx}, \text{ctx}, \Rightarrow_{cx} \rangle$,

We have that $s_{0s} = \langle A_0, m_s, \delta \rangle$ and $s_{0cx} = \langle A_0, m_{cx}, C_0, \delta' \rangle$ where $m_s = m_{cx} = \emptyset$. By the definition of κ_{sic} and τ_{sic} , we also have that their initial ABoxes are the same, and $\delta' = \kappa_{\text{sic}}(\delta)$. Hence, by Lemma 7.45, we have $s_{0s} \sim_{\text{so}} s_{0cx}$. Therefore, by the definition of S-bisimulation, we have $\Upsilon_{\mathcal{G}}^{fs} \sim_{\text{so}} \Upsilon_{\tau_{\text{sic}}(\mathcal{G})}^{f_{\text{B}}^{cx}}$. □

We close this section by showing that the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ properties over S-GKAB can be recast as verification over B-CSGKAB as follows.

Theorem 7.47. *Verification of closed $\mu\mathcal{L}_A^{\text{EQL}}$ properties over S-GKABs can be recast as verification over B-CSGKABs.*

Proof. By Lemma 7.46, we have that $\Upsilon_{\mathcal{G}}^{fs} \sim_{\text{so}} \Upsilon_{\tau_{\text{sic}}(\mathcal{G})}^{f_{\text{B}}^{cx}}$. Hence, by Lemma 5.42 (but consider that now it is between a KB transition system and a context-sensitive transition system), for every $\mu\mathcal{L}_A^{\text{EQL}}$ property Φ , we have that

$$\Upsilon_{\mathcal{G}}^{fs} \models \Phi \text{ if and only if } \Upsilon_{\tau_{\text{sic}}(\mathcal{G})}^{f_{\text{B}}^{cx}} \models \Phi$$

Hence, by using the translation τ_{sic} we can easily transform an S-GKAB into a B-CSGKAB and then the claim easily follows due to the fact above. \square

ALTERNATING GOLOG-KABs

In Chapter 7, we have seen the I-CSGKABs which essentially capture the manipulation of knowledge bases by actions while also taking into account the contextual information and having a mechanism to handle inconsistency. Additionally, we have also seen how the problem of verifying sophisticated temporal properties over I-CSGKABs can be tackled. Basically, we solve that problem by reducing it into the problem of verifying temporal properties over S-GKABs which has been discussed in Chapter 4.

In I-CSGKABs, all computations of the successor states are encapsulated in a single transition that consists of:

1. the action execution which changes the ABox (add or delete facts) which might involve service calls,
2. the context changes, and
3. the inconsistency handling mechanism.

As one might observed, there are several non-determinism sources while computing the successor states of a state that cause several non-deterministic transition from a state to another states. Those sources of non-determinism are:

1. the choice of grounded actions (that is caused by different action parameters or when the actions are wrapped using non-deterministic choice program construct),
2. The choice of service call results,
3. The choice among all possible new contexts, and
4. The choice of repaired ABoxes when there are several possible repairs (the case of b-repairs).

By wrapping all of those non-determinism sources into a single transition, we basically lose the capability to “quantify” over each source of non-determinism above. Thus we lose the ability to do a fine-grained analysis over the detail structure of I-CSGKABs transition system. We can not check some properties such as “*no matter which action is executed, there exists a service call result in which no matter how the context is changing, there exists a repair that leads us into a certain state that satisfy a certain property*”. To cope with this situation, here we introduce the so called *Alternating Golog-KABs* (AGKABs).

The important aspect of AGKABs is that they separate each source of non-determinism. I.e., AGKABs explicitly present the alternation among all of those sources of non-determinism where each source of non-determinism is captured by a single transition. Thus, AGKABs give us more fine-grained transition system. Furthermore, we can also “quantify” over each non-determinism source. AGKABs can also be considered as a model of four player game where the players responding to other players move.

In this chapter we also tackle the problem of verifying a sophisticated temporal properties based on μ -calculus over AGKABs. We solved the problem by reducing it into the problem of verifying $\mu\mathcal{L}_A$ over S-GKABs.

In the following we use $DL-Lite_A$ for expressing KBs and we also do not distinguish between objects and values (thus we drop attributes). Moreover we make use of a countably infinite set Δ of constants, which intuitively denotes all possible values in the system. Additionally, we consider a finite set of distinguished constants $\Delta_0 \subset \Delta$, and a finite set \mathcal{F} of *function symbols* that represents *service calls*, which abstractly account for the injection of fresh values (constants) from Δ into the system. Additionally, for technical development of this chapter, we fix a set $\mathbb{D} = \{d_1, \dots, d_n\}$ of context dimensions. Each context dimension $d_i \in \mathbb{D}$ has its own tree-shaped finite value domain $\langle Dom(d_i), \prec_{d_i} \rangle$, where $Dom(d_i)$ represents the finite set of domain values, and \prec_{d_i} represents the predecessor relation forming the tree.

8.1 AGKABs Formalism and Execution Semantics

The formalism of AGKABs is similar to CSGKABs. We basically formalize AGKABs as a tuple $\mathcal{G}_A = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$ where $T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C$ are the same as in CSGKABs (see Definition 6.13). In the following, we proceed to define the execution semantics for AGKABs.

To define the execution semantics for AGKABs, we adopt the parametric execution semantics of GKABs (see Chapter 4) in order to be able to elegantly accommodate various inconsistency management mechanism. Similar to CSGKABs, the execution semantics of a AGKABs $\mathcal{G}_A = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$ is given in terms of a possibly infinite-state context-sensitive transition system $\mathcal{T}_{\mathcal{G}_A} = \langle \Delta, T_{cx}, \Sigma, s_0, abox, ctx, \Rightarrow \rangle$ (see Definition 6.15 for the detail of $\mathcal{T}_{\mathcal{G}_A}$ components). However, differently from CSGKABs, the states we consider are tuples of the form $\langle id, A, m, C, \delta \rangle$, where id is a unique identifier for the state, A is an ABox, m is a service call map, C is a context and δ is a program. Furthermore, we distinguish the types of states in the transition systems of AGKABs, namely *stable* and *intermediate* states. Technically, we partition the set Σ of the states in $\mathcal{T}_{\mathcal{G}_A}$ into the set Σ_{st} of stable states and the set Σ_{im} of intermediate states (I.e., $\Sigma = \Sigma_{st} \uplus \Sigma_{im}$). Those states in Σ_{st} are called *stable states*, while those in Σ_{im} are called *intermediate states*.

Stable and
Intermediate States

Before we introduce the construction of AGKABs transition systems, in the following we define the notion of AGKABs program execution relation by refining the notion of context-sensitive program execution relation that has been defined in Definition 6.21.

AGKAB Program
Execution Relation

Definition 8.1 (AGKAB Program Execution Relation). Given an AGKAB $\mathcal{G}_A = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$, and a context-sensitive filter relation f^{cx} (see Definition 6.18), we define an *AGKAB program execution relation* $\xrightarrow{\alpha\sigma, f^{cx}}$ as follows:

1. $\langle A, m, C, \mathbf{pick} \langle Q(\vec{p}), \varphi_C \rangle . \alpha(\vec{p}) \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A, m, C, \varepsilon \rangle$,
if the following hold:
 - a) σ is a legal parameter assignment for α in A w.r.t. context C and action invocation $\mathbf{pick} \langle Q(\vec{p}), \varphi_C \rangle . \alpha(\vec{p})$,
 - b) $C \cup \Phi_{\mathbb{D}} \models \varphi_C$.
2. $\langle A, m, C, \delta_1 | \delta_2 \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A, m, C, \delta' \rangle$,
if $\langle A, m, C, \delta_1 \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A, m, C, \delta' \rangle$ or $\langle A, m, \delta_2 \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A, m, C, \delta' \rangle$;

3. $\langle A, m, C, \delta_1; \delta_2 \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A, m, C, \delta'_1; \delta_2 \rangle,$
 if $\langle A, m, C, \delta_1 \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A, m, C, \delta'_1 \rangle;$
4. $\langle A, m, C, \delta_1; \delta_2 \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A, m, C, \delta'_2 \rangle,$
 if $\langle A, m, C, \delta_1 \rangle \in \mathbb{F}$, and $\langle A, m, C, \delta_2 \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A, m, C, \delta'_2 \rangle;$
5. $\langle A, m, C, \text{if } \varphi \text{ then } \delta_1 \text{ else } \delta_2 \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A, m, C, \delta'_1 \rangle,$
 if $\text{ASK}(\varphi, T, A) = \text{true}$, and $\langle A, m, C, \delta_1 \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A, m, C, \delta'_1 \rangle;$
6. $\langle A, m, C, \text{if } \varphi \text{ then } \delta_1 \text{ else } \delta_2 \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A, m, C, \delta'_2 \rangle,$
 if $\text{ASK}(\varphi, T, A) = \text{false}$, and $\langle A, m, C, \delta_2 \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A, m, C, \delta'_2 \rangle;$
7. $\langle A, m, C, \text{while } \varphi \text{ do } \delta \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A, m, C, \delta'; \text{while } \varphi \text{ do } \delta \rangle,$
 if $\text{ASK}(\varphi, T, A) = \text{true}$, and $\langle A, m, C, \delta \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A, m, C, \delta' \rangle.$

■

Now, we proceed to define the construction of AGKABs transition systems that is parameterized with the filter relation f^{cx} as follows.

Definition 8.2 (AGKABs Transition System). Given an AGKAB $\mathcal{G}_A = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$, and a context-sensitive filter relation f^{cx} (see Definition 6.18), we define the *transition system of \mathcal{G}_A w.r.t. f^{cx}* , written $\Upsilon_{\mathcal{G}_A}^{f^{cx}}$, as $\langle \Delta, T_{cx}, \Sigma, s_0, abox, ctx, \Rightarrow \rangle$, where

- $s_0 = \langle id_0, A_0, \emptyset, C_0, \delta \rangle$,
- $\Sigma = \Sigma_{st} \uplus \Sigma_{im}$, and
- Σ_{st} , Σ_{im} and \Rightarrow are defined by simultaneous induction as the smallest sets such that
 - $s_0 \in \Sigma_{st}$, and
 - if $\langle id, A, m, C, \delta \rangle \in \Sigma_{st}$ then we do the following
 - (1) for any action α , and any substitution σ , let

$$S^1 = \{ \langle id^1, A, m, C, \delta' \rangle \mid id^1 \text{ is a fresh identifier, and } \langle A, m, C, \delta \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A, m, C, \delta' \rangle \}$$

- (2) and then for each $s_1 = \langle id^1, A, m, C, \delta' \rangle \in S^1$ with $\langle A, m, C, \delta \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A, m, C, \delta' \rangle$, we do the following:
 - (a) let

$$S^2 = \{ \langle id^2, A, m', C, \delta' \rangle \mid id^2 \text{ is a fresh identifier, } \theta \in \text{EVAL}(\text{ADD}(T_{cx}^C, A, \alpha\sigma)), \text{ and } m' = m \cup \theta \}$$

- (b) and then for each $s_2 = \langle id^2, A, m', C, \delta' \rangle \in S^2$ with $m' = m \cup \theta$, we do the following:
 - (i) let

$$S^3 = \{ \langle id^3, A, m', C', \delta' \rangle \mid id^3 \text{ is a fresh identifier, and } \langle A, C, C' \rangle \in \text{CTX-CHG} \}$$

- (ii) and then for each $s_3 = \langle id^3, A, m', C', \delta' \rangle \in S^3$ with $\langle A, C, C' \rangle \in \text{CTX-CHG}$, we do the following:
if there exists A' such that

$$\langle A, \text{ADD}(T_{cx}^C, A, \alpha\sigma)\theta, \text{DEL}(T_{cx}^C, A, \alpha\sigma), C', A' \rangle \in f^{cx},$$

and A' is $T_{cx}^{C'}$ -consistent. Then for each A' , we have

- * if there exists $\langle id_s, A_s, m_s, C_s, \delta_s \rangle \in \Sigma_{st}$ such that $A_s = A', m_s = m', C_s = C', \delta_s = \delta'$, then

$$s_3 \Rightarrow \langle id_s, A_s, m_s, C_s, \delta_s \rangle,$$

- * otherwise we have $s_4 = \langle id^4, A', m', C', \delta' \rangle \in \Sigma_{st}$, and $s_3 \Rightarrow s_4$, where id^4 is a fresh identifier.

Additionally, we also have that $s_1 \in \Sigma_{im}, s_2 \in \Sigma_{im}, s_3 \in \Sigma_{im}$, as well as $\langle id, A, m, C, \delta \rangle \Rightarrow s_1, s_1 \Rightarrow s_2$, and $s_2 \Rightarrow s_3$. ■

Notice that we never add the states and the transitions that do not lead into a stable state. We use id in each state in order to enforce that the intermediate states between two stable states are unique. This is important in order to prevent false reachability among two stable states. This false reachability could be happened if we do not distinguish two intermediate states that could lead into two different stable states due to the fact that they came from different stable states (Notice that the computation of transition within intermediate states might require some information from its predecessor states).

Similar to Section 7.1, by exploiting the filter relations f_B^{cx}, f_C^{cx} , and f_E^{cx} (see Definitions 7.1, 7.4 and 7.7), we could obtain various execution semantics for AGKABs with different mechanism to handle inconsistencies. Now, employing the b-repair context-sensitive filter f_B^{cx} into AGKABs gives us B-AGKABs that is an AGKABs with *b-repair execution semantics*, i.e., where inconsistent ABoxes are repaired by non-deterministically picking a b-repair. Formally, we define the transition system which provide the b-repair execution semantics for AGKABs as follows.

AGKAB B-Transition
System

Definition 8.3 (AGKAB B-Transition System). Given an AGKAB \mathcal{G}_A and a b-repair filter f_B^{cx} , the *b-transition system* of \mathcal{G}_A , written $\Upsilon_{\mathcal{G}_A}^{f_B^{cx}}$, is the transition system of \mathcal{G}_A w.r.t. f_B^{cx} (see also Definition 8.2). ■

We call *B-AGKABs* the AGKABs adopting this semantics.

By utilizing the c-repair context-sensitive filter f_C^{cx} , we obtain the *c-repair execution semantics* for AGKABs, where inconsistent ABoxes are repaired by computing their unique c-repair. The transition systems which provide the c-repair execution semantics for AGKABs is then defined as follows.

AGKAB C-Transition
System

Definition 8.4 (AGKAB C-Transition System). Given an AGKAB \mathcal{G}_A and a c-repair filter f_C^{cx} , the *c-transition system* of \mathcal{G}_A , written $\Upsilon_{\mathcal{G}_A}^{f_C^{cx}}$, is the transition system of \mathcal{G}_A w.r.t. f_C^{cx} (see also Definition 8.2). ■

We call *C-AGKABs* the AGKABs adopting this semantics.

We now proceed to incorporate the b-evol context-sensitive filter f_E^{cx} into AGKABs that leads us into the *b-evol execution semantics* for AGKABs, where for updates leading to inconsistent ABoxes, their unique bold-evolution is computed. Basically, we define the transition systems which provide the b-evol execution semantics for AGKABs as follows.

Definition 8.5 (AGKAB E-Transition System). Given an AGKAB \mathcal{G}_A and an e-repair filter f_E^{cx} , the *e-transition system* of \mathcal{G}_A , written $\mathcal{T}_{\mathcal{G}_A}^{f_E^{cx}}$, is the transition system of \mathcal{G}_A w.r.t. f_E^{cx} . ■

AGKAB E-Transition System

We call *E-AGKABs* the AGKABs adopting this semantics. Notice that by employing f_E^{cx} in the transition system of AGKABs, we basically assume that the new ABox assertions are consistent with the TBox under the new context (i.e., after the context change). This means that we determine whether the updates are applicable or not based on the new context.

Example 8.6. Recall our simple order processing scenario in Example 6.5. Consider the same context dimensions as in Example 6.5. To model such scenario we specify an AGKAB $\mathcal{G}_A = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$ where T_{cx} , A_0 , Γ , δ , C_0 , and Π_C are the same as in Example 6.14. To give the intuition on how AGKABs are executed, here we provide an example of B-AGKABs execution. Now, let \mathcal{G}_A be a B-AGKAB. The initial state of $\mathcal{T}_{\mathcal{G}_A}^{f_E^{cx}}$ is $s_0 = \langle id_0, A_0, m_0, C_0, \delta \rangle$, where

- id_0 is the unique identifier for this state,
- $A_0 = \{\text{ReceivedOrder}(\text{chair}), \text{ApprovedOrder}(\text{table})\}$,
- $m_0 = \emptyset$,
- $C_0 = \{[PP \rightsquigarrow N], [S \rightsquigarrow NS]\}$,
- $\delta = \mathbf{while} \exists x. [\text{Order}(x)] \wedge \neg [\text{DeliveredOrder}(x)] \mathbf{do} \delta_0$ with
 - 1) $\delta_0 = \delta_1; \delta_2; \delta_3; \delta_4; \delta_5$
 - 2) $\delta_1 = \mathbf{if} \neg [\exists x. \text{ApprovedOrder}(x)]$
 then pick $\langle \text{ReceivedOrder}(x), [PP \rightsquigarrow AP] \wedge [S \rightsquigarrow AS] \rangle. \text{approveOrder}(x)$
 else ε ,
 - 3) $\delta_2 = \mathbf{pick} \langle \text{true}, [PP \rightsquigarrow AP] \wedge [S \rightsquigarrow AS] \rangle. \text{prepareOrders}()$,
 - 4) $\delta_3 = \mathbf{pick} \langle \text{true}, [PP \rightsquigarrow AP] \wedge [S \rightsquigarrow AS] \rangle. \text{assembleOrders}()$,
 - 5) $\delta_4 = \mathbf{pick} \langle \text{true}, \neg ([PP \rightsquigarrow RE] \vee [S \rightsquigarrow PS]) \rangle. \text{checkAssembledOrders}() \mid$
 pick $\langle \text{true}, [PP \rightsquigarrow RE] \vee [S \rightsquigarrow PS] \rangle. \text{outsourceQualityCheck}()$,
 - 6) $\delta_5 = \mathbf{pick} \langle \text{true}, [PP \rightsquigarrow AP] \wedge [S \rightsquigarrow AS] \rangle. \text{deliverOrder}()$.

A snapshot of one possible run in $\mathcal{T}_{\mathcal{G}_A}^{f_E^{cx}}$ is as follows:

$$s_0 \Rightarrow s_1 \Rightarrow s_2 \Rightarrow s_3 \Rightarrow s_4 \Rightarrow \dots$$

where

- $s_1 = \langle id_1, A_0, m_0, C_0, \delta' \rangle$, $s_2 = \langle id_2, A_0, m', C_0, \delta' \rangle$, $s_3 = \langle id_3, A_0, m', C', \delta' \rangle$, and $s_4 = \langle id_4, A', m', C', \delta' \rangle$,
- id_1, id_2, id_3 and id_4 are fresh unique identifiers,
- $\delta' = \delta_3; \delta_4; \delta_5; \mathbf{while} \exists x. [\text{Order}(x)] \wedge \neg [\text{DeliveredOrder}(x)] \mathbf{do} \delta_0$
- $m' = \{[\text{GETDESIGNER}(\text{table}) \rightarrow \text{alice}], [\text{GETDESIGN}(\text{table}) \rightarrow \text{ecodesign}], [\text{ASSIGNASSEMBLINGLOC}(\text{table}) \rightarrow \text{bolzano}]\}$.

- $C' = \{[PP \rightsquigarrow WE], [S \rightsquigarrow PS]\}$ (obtained from the application of context-evolution rule $\langle \text{true}, [PP \rightsquigarrow N] \wedge [S \rightsquigarrow NS] \rangle \mapsto \{[PP \rightsquigarrow WE], [S \rightsquigarrow PS]\}$).
- $A' = \{ \text{ReceivedOrder}(\text{chair}), \text{ApprovedOrder}(\text{table}), \text{designedBy}(\text{table}, \text{alice}), \text{Designer}(\text{alice}), \text{hasDesign}(\text{table}, \text{ecodesign}), \text{hasAssemblingLoc}(\text{table}, \text{bolzano}) \}$

8.2 Verification of AGKABs

The interesting task on AGKABs is to verify whether the evolution of AGKABs satisfy some temporal properties. In this section, we explain the temporal properties formalisms, namely alternating context-sensitive temporal logic $\mu\mathcal{L}_{\text{CTX}}^{\text{Alt}}$, that will be used to specify the temporal properties to be verified over AGKABs. Moreover, we also exhibit the way how we solve the verification problem. The problem definition of the $\mu\mathcal{L}_{\text{CTX}}^{\text{Alt}}$ formula verification over B-AGKABs, C-AGKABs, and E-AGKABs is defined similarly as in CSGKABs (see Definition 6.27). Here we solve this problem by compiling them into S-GKABs and show that the verification of $\mu\mathcal{L}_{\text{CTX}}^{\text{Alt}}$ formulas over B-AGKABs, C-AGKABs, and E-AGKABs can be recast as verification of $\mu\mathcal{L}_A^{\text{EQL}}$ over S-GKABs.

As the verification formalisms, we here we introduce the alternating context-sensitive temporal logic $\mu\mathcal{L}_{\text{CTX}}^{\text{Alt}}$, which is a fragment of $\mu\mathcal{L}_{\text{CTX}}$. The syntax of $\mu\mathcal{L}_{\text{CTX}}^{\text{Alt}}$ is then defined as follows:

Syntax of $\mu\mathcal{L}_{\text{CTX}}^{\text{Alt}}$

$$\begin{aligned} \Phi &:= Q \mid \varphi_C \mid \neg\Phi \mid \Phi_1 \vee \Phi_2 \mid \exists x. \Phi \mid Z \mid \mu Z. \Phi \mid \\ &\quad \langle \rangle \langle \rangle \langle \rangle \langle \rangle \Phi \mid \langle \rangle \langle \rangle \langle \rangle [\] \Phi \mid \langle \rangle \langle \rangle [\] \langle \rangle \Phi \mid \langle \rangle \langle \rangle [\] [\] \Phi \mid \\ &\quad \langle \rangle [\] \langle \rangle \langle \rangle \Phi \mid \langle \rangle [\] \langle \rangle [\] \Phi \mid \langle \rangle [\] [\] \langle \rangle \Phi \mid \langle \rangle [\] [\] [\] \Phi \end{aligned}$$

Essentially, $\mu\mathcal{L}_{\text{CTX}}^{\text{Alt}}$ is a fragment of $\mu\mathcal{L}_{\text{CTX}}$ that is obtained by quadruplicating the modal operator so that we can quantify over all possible sources of non-determinism. Besides the standard abbreviation, we also use the following abbreviation:

$$\begin{aligned} [\] [\] [\] [\] \Phi &= \neg \langle \rangle \langle \rangle \langle \rangle \langle \rangle \neg \Phi, \quad [\] [\] [\] \langle \rangle \Phi = \neg \langle \rangle \langle \rangle \langle \rangle [\] \neg \Phi, \\ [\] [\] \langle \rangle [\] \Phi &= \neg \langle \rangle \langle \rangle [\] \langle \rangle \neg \Phi, \quad [\] [\] \langle \rangle \langle \rangle \Phi = \neg \langle \rangle \langle \rangle [\] [\] \neg \Phi, \\ [\] \langle \rangle [\] [\] \Phi &= \neg \langle \rangle [\] \langle \rangle \langle \rangle \neg \Phi, \quad [\] \langle \rangle [\] \langle \rangle \Phi = \neg \langle \rangle [\] \langle \rangle [\] \neg \Phi, \\ [\] \langle \rangle \langle \rangle [\] \Phi &= \neg \langle \rangle [\] [\] \langle \rangle \neg \Phi, \quad [\] \langle \rangle \langle \rangle \langle \rangle \Phi = \neg \langle \rangle [\] [\] [\] \neg \Phi. \end{aligned}$$

Example 8.7. As an example, the property

$$\begin{aligned} \nu Z. (\forall x. \text{Order}(x) \wedge [S \rightsquigarrow PS] \rightarrow \\ \mu Y. (\text{DeliveredOrder}(x) \vee \langle \rangle [\] [\] [\] Y)) \wedge [\] [\] [\] [\] Z \end{aligned}$$

checks that along every path, it is always true that for every customer order in the peak season, there exists a sequence of action executions leading to the state where the order is delivered, no matter how the service call is evaluated, no matter how the contexts change, and no matter how is the repair behavior

We now proceed to refine the transition systems that provide the execution semantics for AGKABs in such a way that the refined transition systems still provide the same structure except that

- we distinguish the types of the three intermediate states between the two stable states, and
- we provide a mechanism to extract more information from each of the intermediate states (e.g., the corresponding action that was executed before we reach a certain intermediate state).

Hence, it is obvious that the refined version of the AGKAB transition system should satisfy the same $\mu\mathcal{L}_{\text{CTX}}^{\text{Alt}}$ properties.

Towards formalizing the refined transition systems for AGKABs, we first introduce several notions as follows: Given an AGKAB \mathcal{G}_A , let $\mathcal{T}_{\mathcal{G}_A}$ be its context-sensitive transition system. Consider the states s_1, s_2, s_3, s_4 , and s_5 of $\mathcal{T}_{\mathcal{G}_A}$ such that s_1 and s_5 are stable states, s_2, s_3, s_4 are intermediate states, and we have the following transitions:

$$s_1 \Rightarrow s_2 \Rightarrow s_3 \Rightarrow s_4 \Rightarrow s_5$$

By the construction of $\mathcal{T}_{\mathcal{G}_A}$ (see Definition 8.2), it is easy to see that the transitions $s_1 \Rightarrow s_2$ and $s_2 \Rightarrow s_3$ consecutively must be an action execution and an service call evaluation, while the transition $s_3 \Rightarrow s_4$ and $s_4 \Rightarrow s_5$ consecutively must be the context change and the filter application. Now, let $s_1 = \langle id, A, m, C, \delta \rangle$, $s'_2 = \langle id, A, m, C, \delta' \rangle$, α and σ consecutively be the action and the legal parameter assignment that is involved in the transition $s_1 \Rightarrow s_2$, and θ is the corresponding substitution that is involved in the transition $s_2 \Rightarrow s_3$ (i.e., that replaces the “corresponding service calls in the execution of $\alpha\sigma$ over A ”). In this case, we call α (resp. σ) the *source action* (resp. *source action parameters*) of s_2, s_3 and s_4 (Note that the source action and the source action parameters are only defined for the intermediate states). Additionally, we also say that the set $\text{ADD}(T_{cx}^C, A, \alpha\sigma)\theta$, is the *set of facts to be added* of s_3 , and s_4 , while the set $\text{DEL}(T_{cx}^C, A, \alpha\sigma)$ is the *set of facts to be deleted* of s_3 , and s_4 . Furthermore, we also introduce three more types of states for the refined transition system of AGKABs. Technically, we partition the set of states of the AGKABs transition system into four different set of states namely:

*source action and
source action
parameters*

- (i) the set Σ_{st} of *stable states*,
- (ii) the set Σ_{sc} of *service call evaluation states*,
- (iii) the set Σ_{cx} of *context change states*,
- (iv) the set Σ_{ft} of *filter application states*.

We call *stable state* (resp. *service call evaluation state*) a state that belong to Σ_{st} (resp. Σ_{sc}). Similarly, we say that a state is a *context change state* (resp. *filter application state*) if it belongs to Σ_{cx} (resp. Σ_{ft}). Finally, having all necessary ingredients in hand, below we introduce the notion of *fine-grained context-sensitive transition system* which is a refined transition system that provide the execution semantics of an AGKAB.

Definition 8.8 (Fine-Grained Transition System). A *fine-grained transition system* is a tuple $\mathcal{T}_{\mathcal{G}_{cx}} = \langle \Delta, T_{cx}, \Sigma, s_0, abox, ctx, actsrc, actpar, fa, fd, \Rightarrow \rangle$, where:

*Fine-Grained
Transition System*

1. T_{cx} is a contextualized TBox;
2. $\Sigma = \Sigma_{st} \uplus \Sigma_{sc} \uplus \Sigma_{cx} \uplus \Sigma_{ft}$ is a set of states that is partitioned into four different types of states;
3. $s_0 \in \Sigma_{st}$ is the initial state and belongs to Σ_{st} (i.e., s_0 is a stable state);

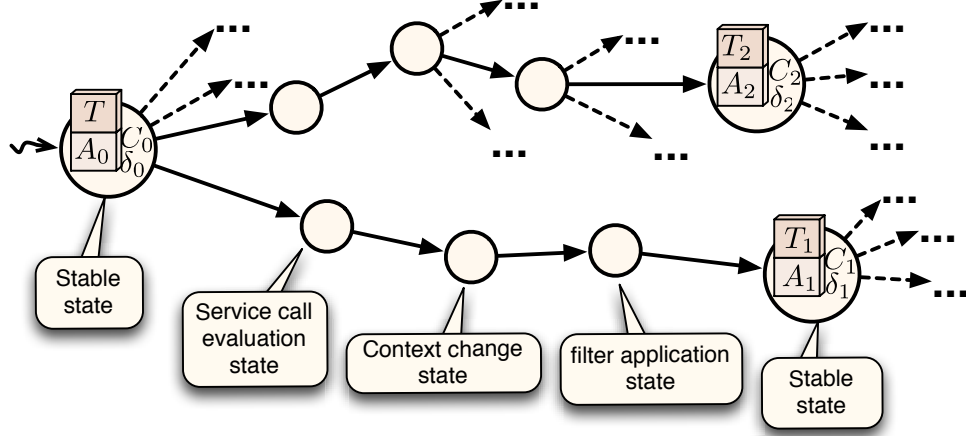


Figure 10: Illustration of fine-grained transition system (Note: T is a TBox, A_i is an ABox, C_i is a context, and δ_i is the remaining program to be executed).

4. $abox$ is a function that, given a state $s \in \Sigma$, returns the ABox associated to s ;
5. ctx is a function that, given a state $s \in \Sigma$, returns the context associated to s ;
6. $actsrc$ is a function that, given a state $s \in \Sigma$, returns the source action name associated to s ;
7. $actpar$ is a function that, given a state $s \in \Sigma$, returns the source action parameters associated to s ;
8. fa is a function that, given a state $s \in \Sigma$, returns the corresponding set of facts to be added associated to s ;
9. fd is a function that, given a state $s \in \Sigma$, returns the corresponding set of facts to be deleted associated to s ;
10. $\Rightarrow \subseteq \Sigma \times \Sigma$ is a transition relation between pairs of states.

Additionally, we require the following:

1. for each $s \in \Sigma_{st}$, if there exists $s' \in \Sigma$ such that $s \Rightarrow s'$, then $s' \in \Sigma_{sc}$.
2. for each $s \in \Sigma_{sc}$, if there exists $s' \in \Sigma$ such that $s \Rightarrow s'$, then $s' \in \Sigma_{cx}$.
3. for each $s \in \Sigma_{cx}$, if there exists $s' \in \Sigma$ such that $s \Rightarrow s'$, then $s' \in \Sigma_{ft}$.
4. for each $s \in \Sigma_{ft}$, if there exists $s' \in \Sigma$ such that $s \Rightarrow s'$, then $s' \in \Sigma_{st}$.

■

Figure 10 illustrates the notion of fine-grained transition system, in particular on the aspect of states alternation.

The construction of a fine-grained transition system of an AGKAB is as follows:

AGKABs
Fine-Grained
Transition System

Definition 8.9 (AGKABs Fine-Grained Transition System). Given an AGKAB $\mathcal{G}_A = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$, and a context-sensitive filter relation f^{cx} (see Definition 6.18), we define the *transition system* of \mathcal{G}_A w.r.t. f^{cx} , written $\Upsilon_{\mathcal{G}_A}^{f^{cx}}$, as $\langle \Delta, T_{cx}, \Sigma, s_0, abox, ctx, actsrc, actpar, fa, fd, \Rightarrow \rangle$, where

- $s_0 = \langle id_0, A_0, \emptyset, C_0, \delta \rangle$, $actsrc(s_0) = fa(s_0) = fd(s_0) = \emptyset$, $actpar(s_0) = \sigma_\emptyset$ (where σ_\emptyset is an empty substitution),
- $\Sigma = \Sigma_{st} \uplus \Sigma_{sc} \uplus \Sigma_{cx} \uplus \Sigma_{ft}$, and
- Σ_{st} , Σ_{sc} , Σ_{cx} , Σ_{ft} and \Rightarrow are defined by simultaneous induction as the smallest sets such that

- $s_0 \in \Sigma_{st}$, and
- if $\langle id, A, m, C, \delta \rangle \in \Sigma_{st}$ then we do the following
 - (1) for any action α , and any substitution σ , let

$$S^1 = \{ \langle id^1, A, m, C, \delta' \rangle \mid id^1 \text{ is a fresh identifier, and} \\ \langle A, m, C, \delta \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A, m, C, \delta' \rangle \}$$

- (2) and then for each $s_1 = \langle id^1, A, m, C, \delta' \rangle \in S^1$ with $\langle A, m, C, \delta \rangle \xrightarrow{\alpha\sigma, f^{cx}} \langle A, m, C, \delta' \rangle$, we do the following:
 - (a) let

$$S^2 = \{ \langle id^2, A, m', C, \delta' \rangle \mid id^2 \text{ is a fresh identifier,} \\ \theta \in \text{EVAL}(\text{ADD}(T_{cx}^C, A, \alpha\sigma)), \\ \text{and } m' = m \cup \theta \}$$

- (b) and then for each $s_2 = \langle id^2, A, m', C, \delta' \rangle \in S^2$ with $m' = m \cup \theta$, we do the following:
 - (i) let

$$S^3 = \{ \langle id^3, A, m', C', \delta' \rangle \mid id^3 \text{ is a fresh identifier, and} \\ \langle A, C, C' \rangle \in \text{CTX-CHG} \}$$

- (ii) and then for each $s_3 = \langle id^3, A, m', C', \delta' \rangle \in S^3$ with $\langle A, C, C' \rangle \in \text{CTX-CHG}$, we do the following:

if there exists A' such that

$$\langle A, \text{ADD}(T_{cx}^C, A, \alpha\sigma)\theta, \text{DEL}(T_{cx}^C, A, \alpha\sigma), C', A' \rangle \in f^{cx},$$

and A' is $T_{cx}^{C'}$ -consistent. Then for each of that A' , we have

- * if there exists $\langle id_s, A_s, m_s, C_s, \delta_s \rangle \in \Sigma_{st}$ such that $A_s = A', m_s = m', C_s = C', \delta_s = \delta'$, then we have $s_3 \Rightarrow \langle id_s, A_s, m_s, C_s, \delta_s \rangle$,
 - * otherwise we have $s_4 = \langle id^4, A', m', C', \delta' \rangle \in \Sigma_{st}$, where id^4 is a fresh identifier, and we have $s_3 \Rightarrow s_4$.
- Additionally, we also have that $s_1 \in \Sigma_{sc}$, $s_2 \in \Sigma_{cx}$, $s_3 \in \Sigma_{ft}$, $\langle id, A, m, C, \delta \rangle \Rightarrow s_1$, $s_1 \Rightarrow s_2$, $s_2 \Rightarrow s_3$, and
- * $actsrc(s_1) = actsrc(s_2) = actsrc(s_3) = \alpha$,
 - * $actsrc(s_4) = \emptyset$
 - * $actpar(s_1) = actpar(s_2) = actpar(s_3) = \sigma$,
 - * $actpar(s_4) = \sigma_\emptyset$ (where σ_\emptyset is an empty substitution),
 - * $fa(s_1)$ = the set of ABox assertions in $\text{ADD}(T_{cx}^C, A, \alpha\sigma)$ (excluding the ground skolem terms).
 - * $fa(s_2) = fa(s_3) = \text{ADD}(T_{cx}^C, A, \alpha\sigma)\theta$,
 - * $fd(s_1) = fd(s_2) = fd(s_3) = \text{DEL}(T_{cx}^C, A, \alpha\sigma)$,
 - * $fa(s_4) = fd(s_4) = \emptyset$,

■

Notice that in the construction above we do not define the information of *actpar*, *fa*, and *fd* for the stable states. The reason is because a stable state might be reached from more than one different run, hence it might be obtained through various different way of manipulating states (e.g., different executed action, different fact to be added, and different facts to be deleted).

From this moment, unless explicitly stated differently, we consider that the execution semantics of AGKABs is given by fine-grained transition system.

Lemma 8.10. *Given an AGKAB \mathcal{G}_A , a context-sensitive filter relation f^{cx} , and a $\mu\mathcal{L}_{\text{CTX}}^{\text{Alt}}$ formula Φ . Let Υ_1 be the context-sensitive transition system of \mathcal{G}_A w.r.t. f^{cx} as in Definition 8.2, and Υ_2 be the fine-grained transition system of \mathcal{G}_A w.r.t. f^{cx} as in Definition 8.9, we have that*

$$\Upsilon_1 \models \Phi \text{ if and only if } \Upsilon_2 \models \Phi$$

Proof. The claim easily follows from Definitions 8.2 and 8.9 since essentially both transition systems have the same structure except that the fine-grained transition systems provide more capabilities to access the information within a state. \square

As the next preliminaries, in this chapter we reserve several fresh concept/role names as follows:

- In order to mimic the context evolution within S-GKABs, we simply adopt our approach in Section 6.5.2.1. Thus, similar to Section 6.5.2.1, for each context dimension assignment $[d_i \rightsquigarrow v_j]$ we reserve two fresh concept names $D_i^{v_j}$ and $\mathbf{D}_i^{v_j}$ in order to represent it as an ABox assertion. Similarly, this kind of concept name is also called *context dimension concept*.
- Given any contextualized TBox T_{cx} , for each concept name $N \in \text{voc}(T_{cx})$, we reserve two fresh concept names N^a and N^d to keep track the temporary information about ABox assertions to be added/deleted before we materialize the update (similarly for role names). The concept names of the form N^a (resp. N^d) is called *added* (resp. *deleted*) *fact marker* concept names (similarly for role names, we call them *added* (resp. *deleted*) *fact marker* role names).
- To keep track the information of the source action, we reserve a fresh concept name *Actsrc* and it will be populated only with special constants that is reserved to represent action names. Therefore, w.l.o.g., here we assume that each action name is unique. Note that we can easily enforce this assumption by renaming each action name that occurs more than once.
- Given any action α with parameters p_1, \dots, p_m , we reserve m fresh concept names Par_1, \dots, Par_m (similarly for role names). This kind of concept names are called *action parameter concept names*. Therefore, w.l.o.g., we also assume that each action parameter has a unique name and each of them has a corresponding reserved concept name. Note that we can easily enforce this situation by renaming each action parameter name that occurs more than once.
- To introduce several types of states for KB transition system, we reserve several ABox assertions namely $\text{St}(\text{servCl})$, $\text{St}(\text{ctxChg})$, $\text{St}(\text{filt})$, and $\text{St}(\text{int})$.

Here, we call *special marker* the concept/role assertions made by using the reserved concept/role names above. Related to the reserved concept/role names above, we introduce the notion of a set of added/deleted assertions as follows.

Definition 8.11 (Added Assertions). Given a contextualized TBox T_{cx} , and an ABox A over $\text{VOC}(T_{cx})$, we define the set of *added assertions of A* as a set $\text{ADD}(A) \subseteq A$ of ABox assertions such that we have concept assertion $N^a(c) \in \text{ADD}(A)$ if and only if $N^a(c) \in A$ and N^a is an added fact marker concept name (similarly for role assertions). ■

Added Assertions

Definition 8.12 (Deleted Assertions). Given a contextualized TBox T_{cx} , and an ABox A over $\text{VOC}(T_{cx})$, we define the set of *deleted assertions of A* as a set $\text{DEL}(A) \subseteq A$ of ABox assertions such that we have concept assertion $N^d(c) \in \text{DEL}(A)$ if and only if $N^d(c) \in A$ and N^d is a deleted fact marker concept name (similarly for role assertions). ■

Deleted Assertions

Furthermore, we also introduce the notion of a set of action parameter assertions and a set of context assertions as follows.

Definition 8.13 (Action Parameter Assertions). Given a contextualized TBox T_{cx} , and an ABox A over $\text{VOC}(T_{cx})$, we define the set of *action parameter assertions of A* as a set $\text{PAR}(A) \subseteq A$ of ABox assertions such that we have concept assertion $Par(c) \in \text{PAR}(A)$ if and only if $Par(c) \in A$ and Par is an action parameter concept name. ■

Action Parameter Assertions

Definition 8.14 (Context Assertions). Given a contextualized TBox T_{cx} , and an ABox A over $\text{VOC}(T_{cx})$, we define the set of *context assertions of A* as a set $\text{CTX}(A) \subseteq A$ of ABox assertions such that we have concept assertion $D_i^{v_j}(c) \in \text{CTX}(A)$ if and only if $D_i^{v_j}(c) \in A$ and $D_i^{v_j}$ is a context dimension concept name. ■

Context Assertions

We now proceed to introduce specific fragment of KB transition systems namely *typed KB transition systems*. The idea is that they will be the transition systems of S-GKABs that mimics the evolution of AGKABs that is provided by the fine-grained transition systems. As preliminaries, we now introduce several types of states namely *stable state*, *service call evaluation state*, *context change state*, *filter application state*, and *filter application intermediate state*. All of them are formally defined below:

Definition 8.15 (Stable State). Let \mathcal{T} be a KB transition system, a state s of \mathcal{T} is called a *stable state* if $\text{St}(\text{servCl}) \notin \text{abox}(s)$, $\text{St}(\text{ctxChg}) \notin \text{abox}(s)$, $\text{St}(\text{filt}) \notin \text{abox}(s)$, $\text{St}(\text{int}) \notin \text{abox}(s)$. ■

Stable State

Definition 8.16 (Service Call Evaluation State). Let \mathcal{T} be a KB transition system, a state s of \mathcal{T} is called a *service call evaluation state* if $\text{St}(\text{servCl}) \in \text{abox}(s)$, $\text{St}(\text{ctxChg}) \notin \text{abox}(s)$, $\text{St}(\text{filt}) \notin \text{abox}(s)$, $\text{St}(\text{int}) \notin \text{abox}(s)$. ■

Service Call Evaluation State

Definition 8.17 (Context Change State). Let \mathcal{T} be a KB transition system, a state s of \mathcal{T} is called a *context change state* if $\text{St}(\text{servCl}) \notin \text{abox}(s)$, $\text{St}(\text{ctxChg}) \in \text{abox}(s)$, $\text{St}(\text{filt}) \notin \text{abox}(s)$, $\text{St}(\text{int}) \notin \text{abox}(s)$. ■

Context Change State

Definition 8.18 (Filter Application State). Let \mathcal{T} be a KB transition system, a state s of \mathcal{T} is called a *filter application state* if $\text{St}(\text{servCl}) \notin \text{abox}(s)$, $\text{St}(\text{ctxChg}) \notin \text{abox}(s)$, $\text{St}(\text{filt}) \in \text{abox}(s)$, $\text{St}(\text{int}) \notin \text{abox}(s)$. ■

Filter Application State

Filter Application
Intermediate State

Definition 8.19 (Filter Application Intermediate State). Let \mathcal{T} be a KB transition system, a state s of \mathcal{T} is called a *filter application state* if $\text{St}(\text{servCl}) \notin \text{abox}(s)$, $\text{St}(\text{ctxChg}) \notin \text{abox}(s)$, $\text{St}(\text{filt}) \notin \text{abox}(s)$, $\text{St}(\text{int}) \in \text{abox}(s)$. ■

Notice that all of the types introduced above, except the filter application intermediate states, are the same as the types of states in the fine-grained transition system. The important different is that for KB transition system, the type is determined by the special ABox assertion that is contained by the state.

Having the necessary ingredients in hand, we are now ready to define the notion of typed KB transition system as follows.

Typed KB Transition
System

Definition 8.20 (Typed KB Transition System). Given a KB transition system $\mathcal{T} = \langle \Delta, T, \Sigma, s_0, \text{abox}, \Rightarrow \rangle$ we say that \mathcal{T} is a *typed KB transition system* if the following hold:

- s_0 is a stable state,
- for each state $s \in \Sigma$, we have the following:
 - s is either a stable state (see Definition 8.15), a service call evaluation state (see Definition 8.16), a context change state (see Definition 8.17), a filter application state (see Definition 8.18), or a filter application intermediate state (see Definition 8.19),
 - if s is a stable state, and there exists $s' \in \Sigma$ such that $s \Rightarrow s'$, then s' is a service call evaluation state.
 - if s is a service call evaluation state, and there exists $s' \in \Sigma$ such that $s \Rightarrow s'$, then s' is a context change state.
 - if s is a context change state, and there exists $s' \in \Sigma$ such that $s \Rightarrow s'$, then s' is a filter application state.
 - if s is a filter application state, and there exists $s' \in \Sigma$ such that $s \Rightarrow s'$, then s' is a filter application intermediate state.
 - if s is a filter application intermediate state, and there exists $s' \in \Sigma$ such that $s \Rightarrow s'$, then s' is either a filter application intermediate state or a stable state.

■

Intuitively, compare to the ordinary KB transition systems, the Typed KB transition systems require that the states must be either a stable state, service call evaluation state, context change state, filter application state, and filter application intermediate state. Additionally, the Typed KB transition systems require that there is an alternation of state types among the transitions of the states. In particular, the order of alternation should be as follows: (i) a stable state should be followed by a service call evaluation state (ii) and then a service call evaluation state should be followed by a context change state, (iii) after that the successor of a context change state should be a filter application state (iv) then a filter application state should be continued by filter application intermediate state. (v) last, the filter application intermediate state should be followed by either a filter application intermediate state or a stable state.

8.2.1 Verification of B-AGKABs

This section is aimed to present the reduction of the $\mu\mathcal{L}_{\text{CTX}}^{\text{Alt}}$ verification over B-AGKABs into the $\mu\mathcal{L}_A^{\text{EQL}}$ verification over S-GKABs. We first explain how we transform B-AGKABs into S-GKABs, and also how we translate $\mu\mathcal{L}_{\text{CTX}}^{\text{Alt}}$ formulas into $\mu\mathcal{L}_A^{\text{EQL}}$ formulas. Then, we introduce a specific notion of bisimulation that will be used to prove our reduction from B-AGKABs into S-GKABs. Last, we show that we can recast the verification of B-AGKABs into S-GKABs.

8.2.1.1 Translating B-AGKABs to S-GKABs

We devote this section to explain our translation that transforms B-AGKABs into S-GKABs. To do so, several preliminaries need to be introduced first.

As it can be seen from the execution semantics of AGKABs, they separate the service call evaluation from the action execution. Moreover, the ABox is not directly updated with the changes that is done by the action. Thus, to simulate that situation inside S-GKABs, in the following we introduce the notion of splitting an action as follows:

Definition 8.21 (Split Action). Given an action $\alpha(p_1, \dots, p_m) : \{e_1, \dots, e_n\}$ with $e_i = [q_i^+] \wedge Q_i^- \rightsquigarrow \mathbf{add} F_i^+, \mathbf{del} F_i^-$. Let Par_1, \dots, Par_m be the fresh concept names for capturing the value of each action parameter, the *split actions* obtained from α are two actions α_a and α_b as follows:

1. α_a is a fresh action name and is of the form $\alpha_a(p_1, \dots, p_m) : \{e'_1, \dots, e'_n, e_{\text{temp}}, e_{\text{par}}, e_{\text{act}}\}$, where
 - e'_i (for $i \in \{1, \dots, n\}$) is obtained from e_i such that $e'_i = Q_{cx}^i \rightsquigarrow \mathbf{add} F_i^{+'} \cup F_i^{-'}$ where:
 - Q_{cx}^i is a contextually compiled query of $[q_i^+] \wedge Q_i^-$ w.r.t. \mathbb{D} .
 - for each atom $N(t) \in F_i^+$, such that t is not a skolem terms (representing a service call), we have $N^a(t) \in F_i^{+'}$.
 - for each atom $P(t_1, t_2) \in F_i^+$, such that t_1 and t_2 are not skolem terms (representing a service call), we have $P^a(t_1, t_2) \in F_i^{+'}$.
 - for each atom $N(t) \in F_i^-$ (resp. $P(t_1, t_2) \in F_i^-$), we have $N^d(t) \in F_i^{-'}$ (resp. $P^d(t_1, t_2) \in F_i^{-'}$).
 - $e_{\text{temp}} = \{\mathbf{true} \rightsquigarrow \mathbf{add} \{\text{St}(\text{servCl})\}\}$
 - $e_{\text{par}} = \{\mathbf{true} \rightsquigarrow \mathbf{add} \{Par_1(p_1), \dots, Par_m(p_m)\}\}$
 - $e_{\text{act}} = \{\mathbf{true} \rightsquigarrow \mathbf{add} \{\text{Actsrc}(\alpha)\}\}$
2. α_b is a fresh action name and it is a 0-ary action of the form $\alpha_b() : \{e'_1, \dots, e'_n, e_{\text{temp}}\}$, where
 - e'_i (for $i \in \{1, \dots, n\}$) is obtained from e_i such that

$$e'_i = Q_{cx}^i \wedge Par_1(p_1) \wedge \dots \wedge Par_m(p_m) \rightsquigarrow \mathbf{add} F_i^{+'}$$

where

- Q_{cx}^i is contextually compiled query of $[q_i^+] \wedge Q_i^-$ w.r.t. \mathbb{D} .
- and $F_i^{+'}$ is obtained as follows:
 - * for each atom $N(t) \in F_i^+$, such that t is a skolem terms (representing a service call), we have $N^a(t) \in F_i^{+'}$.

- * for each atom $P(t_1, t_2) \in F_i^+$, such that t_1 and t_2 are skolem terms (representing a service call), we have $P^a(t_1, t_2) \in F_i^{+'}$.
- $e_{temp} = \{\mathbf{true} \rightsquigarrow \mathbf{add} \{\mathbf{St}(ctxChg)\}, \mathbf{del} \{\mathbf{St}(servCl)\}\}$

In this case we also say that α_a is the *first split action* of α and α_b is the *second split action* of α . ■

The main purpose of introducing split actions is to separate the non-determinism sources that come from the choice of actions and also the choice of service call substitutions. Basically, the split actions of an action are obtained by splitting an action into two actions where one action do additions/deletions that do not involve any service calls, while the other action do additions that involve service calls. Additionally, the split actions of an action α do not concretely add/delete the assertions but only give marks on the assertions that need to be added/deleted by α . This is done by adding concept/role assertions made by the added/deleted fact marker concept/role names. We will see later that the materialization of the update (addition/deletion) is done by update action. As a further intuition, in the second split action, we make use a query that is made by action parameter concept names in order to enforce that it uses the same action parameter as the corresponding first split action.

Since inside a program an action is always appeared as an action invocation, in the following we introduce the notion of split action invocation that is based on the notion of split action.

*Split Action
Invocation*

Definition 8.22 (Split Action Invocation). A *split action invocation* obtained from a context-sensitive atomic action invocation $\mathbf{pick} \langle Q(\vec{p}), \varphi_C \rangle . \alpha(\vec{p})$ is a program

$$\mathbf{pick} Q'(\vec{p}).\alpha_a(\vec{p}); \mathbf{pick} \mathbf{true}.\alpha_b()$$

where

- $Q' = Q_{cx} \wedge q_{\varphi_C}$, where Q_{cx} is a contextually compiled query of Q (see Definition 6.34), and q_{φ_C} is the query that represents the context expression φ_C (see Definition 6.32).
- α_a and α_b are split actions obtained from α (see Definition 8.21). ■

In AGKABs, after evaluating the service call, the next step is changing the context. To mimic this step inside S-GKABs, we make use the context-change program obtained from the set of context evolution rule as introduced in Definition 6.39. However, we can not use it directly since it materializes the updates (additions/deletions). Thus, here we refine the notion of action and action invocation obtained from context-evolution rule (introduced in Definition 6.38) and introduce the notion of sole action and sole action invocation obtained from context-evolution rule. The core different is that in the sole action obtained from context-evolution rule, the action only changes the context and does not concretize the assertions to be added/deleted.

*Sole Action and Sole
Action Invocation
Obtained From
Context-evolution
Rule*

Definition 8.23 (Sole Action and Sole Action Invocation Obtained From Context-evolution Rule). A *sole action invocation obtained from a context-evolution rule* $\langle Q, \varphi_C \rangle \mapsto C_{new}$ in Π_C , is an action invocation $\mathbf{pick} Q'.\alpha_C^s()$ where

1. $Q' = Q_{cx} \wedge q_{\varphi_C}$ where Q_{cx} is contextually compiled query of Q , and q_{φ_C} is the query obtained from the context expression φ_C .

2. α_C^s is a 0-ary action obtained from $\langle Q, \varphi_C \rangle \mapsto C_{new}$ as follows:

- For each $[d_i \mapsto v_j] \in C_{new}$, we have:
 - (i) $\text{true} \rightsquigarrow \mathbf{add} \{D_i^{v_j}(\mathbf{c}), \mathbf{D}_i^{v_j}(\mathbf{c})\}$ in $\text{EFF}(\alpha_C^s)$, and
 - (ii) $\text{true} \rightsquigarrow \mathbf{del} \{D_i^{v_k}(\mathbf{c}), \mathbf{D}_i^{v_k}(\mathbf{c})\}$ in $\text{EFF}(\alpha_C^s)$ for every $v_k \in \text{Dom}(d_i)$ such that $v_k \neq v_j$.
- Additionally, we have

$$\text{true} \rightsquigarrow \mathbf{add} \{\text{St}(\text{filt})\}, \mathbf{del} \{\text{St}(\text{ctxChg})\} \text{ in } \text{EFF}(\alpha_C^s).$$

In this case we say that α_C^s is a sole action obtained from the context-evolution rule $\langle Q, \varphi_C \rangle \mapsto C_{new}$.

■

To concretize the addition/deletion over the ABox, in the following we introduce the notion of update action which essentially materializes the updates.

Definition 8.24 (Update Action). Let $\mathcal{G}_{cx} = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$ be an AGKAB. An *update action* α^u is 0-ary (i.e., has no action parameters) action over T_{cx} , where $\text{EFF}(\alpha^u)$ is the smallest set containing the following effects:

Update Action

- For each concept name $N \in \text{VOC}(T_{cx})$, we have
 - (i) $N^a(x) \rightsquigarrow \mathbf{add} \{N(x)\}, \mathbf{del} \{N^a(x)\}$ in $\text{EFF}(\alpha^u)$, and
 - (ii) $N^d(x) \rightsquigarrow \mathbf{del} \{N(x), N^d(x)\}$ in $\text{EFF}(\alpha^u)$.
- Similarly for the role names, we create the same effect as above.
- $\text{Actsrc}(x) \rightsquigarrow \mathbf{del} \{\text{Actsrc}(x)\}$ in $\text{EFF}(\alpha^u)$
- For each action parameter concept name Par , we have

$$Par(x) \rightsquigarrow \mathbf{del} \{Par(x)\} \text{ in } \text{EFF}(\alpha^u)$$
- Additionally, we have

$$\text{true} \rightsquigarrow \mathbf{add} \{\text{St}(\text{int})\}, \mathbf{del} \{\text{St}(\text{filt})\} \text{ in } \text{EFF}(\alpha^u).$$

■

Basically, the update action simply adds (resp. deletes) the ABox assertions that has been marked to be added (resp. deleted).

As the last preliminary towards introducing our translation that transforms B-AGKABs into S-GKABs, in the following we introduce the notion of program translation for B-AGKABs.

Definition 8.25 (Program Translation κ_B^A). Given a B-AGKABs $\mathcal{G}_A = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$ we define a *translation* κ_B^A which translates a program δ into a program δ' inductively as follows:

Program Translation κ_B^A

$$\begin{aligned}
 \kappa_B^A(\mathbf{pick} \langle Q(\vec{p}), \varphi_C \rangle. \alpha(\vec{p})) &= \mathbf{pick} Q'(\vec{p}).\alpha_a(\vec{p}); \mathbf{pick} \text{true}.\alpha_b(); \delta_{\Pi_C}; \\
 &\quad \mathbf{pick} \text{true}.\alpha^u(); \delta_b^{T_{cx}}; \mathbf{pick} \text{true}.\alpha_{temp}^-() \\
 \kappa_B^A(\varepsilon) &= \varepsilon \\
 \kappa_B^A(\delta_1 | \delta_2) &= \kappa_B^A(\delta_1) | \kappa_B^A(\delta_2) \\
 \kappa_B^A(\delta_1; \delta_2) &= \kappa_B^A(\delta_1); \kappa_B^A(\delta_2) \\
 \kappa_B^A(\mathbf{if} \varphi \mathbf{then} \delta_1 \mathbf{else} \delta_2) &= \mathbf{if} \varphi \mathbf{then} \kappa_B^A(\delta_1) \mathbf{else} \kappa_B^A(\delta_2) \\
 \kappa_B^A(\mathbf{while} \varphi \mathbf{do} \delta) &= \mathbf{while} \varphi \mathbf{do} \kappa_B^A(\delta)
 \end{aligned}$$

where

- **pick** $Q'(\vec{p}).\alpha_a(\vec{p}); \mathbf{pick} \text{ true}.\alpha_b()$ is a split action invocation obtained from **pick** $\langle Q(\vec{p}), \varphi_C \rangle.\alpha(\vec{p})$ as in Definition 8.22,
- δ_{Π_C} is a context-change program obtained from Π_C as in Definition 6.39 except that it is formed by sole action invocation obtained from context evolution rule as in Definition 8.23,
- α^u is an update action (see Definition 8.24),
- $\delta_b^{T_{cx}}$ is a context-sensitive b-repair program over T_{cx} as in Definition 7.12,
- $\alpha_{temp}^-() : \{\mathbf{true} \rightsquigarrow \mathbf{del} \{\mathbf{St}(int)\}\}$.

■

Essentially, the program translation κ_B^A translates each context-sensitive action invocation into a sequence of split action invocation that is obtained from the given action invocation and then concatenates it with the context-change program, the update action and the program that simulates the b-repair computation. Thus, intuitively, it can be seen that the obtained program separate the source of non-determinism as in B-AGKABs.

Having all ingredients in hand, we now proceed to define the translation from B-AGKABs into S-GKABs as follows.

Translation from
B-AGKAB to
S-GKAB

Definition 8.26 (Translation from B-AGKAB to S-GKAB). We define a translation τ_B^A that, given a B-AGKAB $\mathcal{G}_A = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$, produces an S-GKAB $\tau_B^A(\mathcal{G}_A) = \langle T_{\mathbb{D}}, A_0 \cup A_{C_0}, \Gamma', \delta' \rangle$, where

- $T_{\mathbb{D}}$ is a TBox obtained from a set of context dimensions \mathbb{D} (see Definition 6.29),
- A_{C_0} is an ABox obtained from C_0 (see Definition 6.31),
- $\Gamma' = \Gamma_\alpha \cup \Gamma_C \cup \Gamma_b^{T_{cx}} \cup \{\alpha^u, \alpha_{temp}^-\}$ where:
 - Γ_α is obtained from Γ such that for each action $\alpha \in \Gamma$, we have $\alpha_1, \alpha_2 \in \Gamma_\alpha$ where α_1 and α_2 are split action obtained from α (see Definition 8.21),
 - Γ_C is obtained from Π_C such that for each context-evolution rule $\langle Q, \varphi_C \rangle \mapsto C_{new}$ in Π_C , we have $\alpha_C \in \Gamma_C$ where α_C is a sole action obtained from the context-evolution rule $\langle Q, \varphi_C \rangle \mapsto C_{new}$ (see Definition 8.23),
 - $\Gamma_b^{T_{cx}}$ is the set of context-sensitive b-repair actions over T_{cx} (see Definition 7.10),
 - α^u is an update action (see Definition 8.24),
 - α_{temp}^- is an action of the form $\alpha_{temp}^-() : \{\mathbf{true} \rightsquigarrow \mathbf{del} \{\mathbf{St}(int)\}\}$.
- $\delta' = \kappa_B^A(\delta)$.

■

In the following, we formally state that the transition system of the S-GKAB that is obtained from a B-AGKAB is a typed KB transition system.

Lemma 8.27. *Given a B-AGKAB \mathcal{G}_A with transition system $\Upsilon_{\mathcal{G}_A}^{f_{cx}^B}$, let $\tau_B^A(\mathcal{G}_A)$ be the S-GKAB obtained from \mathcal{G}_A through τ_B^A and $\Upsilon_{\tau_B^A(\mathcal{G}_A)}^{f_S}$ be its transition system. We have that $\Upsilon_{\tau_B^A(\mathcal{G}_A)}^{f_S}$ is typed KB transition system.*

Proof. The claim easily follows from the construction of $\tau_B^A(\mathcal{G}_A)$ by also considering the following:

- based on the program translation κ_B^A (see Definition 8.25), especially on the part of atomic invocation translation, it is easy to see that we have an alternation

between stable state, service call evaluation state, context-change state, and filter application state.

- also by the definition of κ_B^A (see Definition 8.25) all states in $\tau_B^A(\mathcal{G}_A)$ are either stable state, service call evaluation state, context-change state, or filter application state.
- the initial state of $\tau_B^A(\mathcal{G}_A)$ is a stable state.

□

The $\mu\mathcal{L}_{\text{CTX}}^{\text{Alt}}$ property Φ over a B-AGKAB \mathcal{G}_A can then be recast as a corresponding $\mu\mathcal{L}_A^{\text{EQL}}$ property over an S-GKAB $\tau_B^A(\mathcal{G}_A)$ using the following formula translation:

Definition 8.28 (Translation t_j^A). We define a *translation* t_j^A that transforms an arbitrary $\mu\mathcal{L}_{\text{CTX}}^{\text{Alt}}$ formula Φ (in NNF) into a $\mu\mathcal{L}_A^{\text{EQL}}$ formula Φ' inductively by recurring over the structure of Φ as follows:

Translation t_j^A

- $t_j^A(Q) = Q_{cx}$
- $t_j^A(\varphi_C) = q_{\varphi_C}$
- $t_j^A(\neg Q) = \neg Q_{cx}$
- $t_j^A(Qx.\Phi) = Qx.t_j^A(\Phi)$
- $t_j^A(\Phi_1 \circ \Phi_2) = t_j^A(\Phi_1) \circ t_j^A(\Phi_2)$
- $t_j^A(\odot Z.\Phi) = \odot Z.t_j^A(\Phi)$
- $t_j^A(\odot \odot \odot \langle \neg \rangle \Phi) =$
 $\odot \odot \odot \langle \neg \rangle \mu Z.((\text{St}(int) \wedge \langle \neg \rangle Z) \vee (\neg \text{St}(int) \wedge t_j^A(\Phi)))$
- $t_j^A(\odot \odot \odot [\neg] \Phi) =$
 $\odot \odot \odot [\neg] \mu Z.((\text{St}(int) \wedge [\neg] Z \wedge \langle \neg \rangle \top) \vee (\neg \text{St}(int) \wedge t_j^A(\Phi)))$

where:

- \circ is a binary operator ($\vee, \wedge, \rightarrow$, or \leftrightarrow),
- \odot is least (μ) or greatest (ν) fix-point operator,
- Q is forall (\forall) or existential (\exists) quantifier.
- \odot is box ($[\neg]$) or diamond ($\langle \neg \rangle$) modal operator.

■

8.2.1.2 Alternating Jumping Bisimulation (AJ-Bisimulation)

As a vehicle to reduce the verification of B-AGKABs into S-GKABs, in this section we introduce the notion of alternating jumping bisimulation (AJ-Bisimulation). Additionally, here we also show an important lemma about the situation where two AJ-bisimilar transition systems can not be distinguished by certain temporal properties.

As a preliminary towards defining the notion of AJ-Bisimulation, below we introduce the notion of equality between states that ignores the special markers and also consider different representations of contextual information.

Definition 8.29 (Contextually Equal Modulo Special Markers).

Let $\mathcal{T}_1 = \langle \Delta, T_{cx}, \Sigma_1, s_{01}, abox_1, ctx, actsrc, actpar, fa, fd, \Rightarrow_1 \rangle$ be a fine-grained transition system, and $\mathcal{T}_2 = \langle \Delta, T, \Sigma_2, s_{02}, abox_2, \Rightarrow_2 \rangle$ be a KB transition system. Consider two states $s_1 \in \Sigma_1$ and $s_2 \in \Sigma_2$, we say s_1 is *contextually equal modulo special markers* to s_2 , written $s_1 \simeq_{cx} s_2$, if the following hold

Contextually Equal Modulo Special Markers

- $\text{voc}(T) = \text{voc}(T_{cx})$,

- For each concept name $N \in \text{voc}(T)$ (i.e., N is not a special marker concept name), we have a concept assertion $N(c) \in A_1$ if and only if a concept assertion $N(c) \in A_2$,
- For each role name $P \in \text{voc}(T)$, we have a role assertion $P(c_1, c_2) \in A_1$ if and only if a role assertion $P(c_1, c_2) \in A_2$.
- We have context ABox assertion $\mathbf{D}(\mathbf{c}) \in \text{abox}_2(s_2)$ if and only if $\mathbf{D}(\mathbf{c}) \in A_{\text{ctx}(s_1)}$ (recall that $A_{\text{ctx}(s_1)}$ is the set of ABox assertions that represents a context as defined in Definition 6.31).

■

The AJ-bisimulation is then defined as follows:

*Alternating Jumping
Bisimulation
(AJ-Bisimulation)*

Definition 8.30 (Alternating Jumping Bisimulation (AJ-Bisimulation)).

Let $\mathcal{T}_1 = \langle \Delta, T_{\text{cx}}, \Sigma_1, s_{01}, \text{abox}_1, \text{ctx}, \text{actsrc}, \text{actpar}, \text{fa}, \text{fd}, \Rightarrow_1 \rangle$ be a fine-grained transition system, and $\mathcal{T}_2 = \langle \Delta, T, \Sigma_2, s_{02}, \text{abox}_2, \Rightarrow_2 \rangle$ be a typed KB transition system, with $\text{ADOM}(\text{abox}_1(s_{01})) \subseteq \Delta$, $\text{ADOM}(\text{abox}_2(s_{02})) \subseteq \Delta$, and s_{01} as well as s_{02} are stable states. An alternating jumping bisimulation (AJ-Bisimulation) between \mathcal{T}_1 and \mathcal{T}_2 is a relation $\mathcal{B} \subseteq \Sigma_1 \times \Sigma_2$ such that $\langle s_1, s_2 \rangle \in \mathcal{B}$ implies that one of the following condition must hold:

1. s_1 as well as s_2 both are either stable states, service call evaluation states, or context change states and we have the following:
 - a) $s_1 \simeq_{\text{cx}} s_2$
 - b) for each s'_1 , if $s_1 \Rightarrow_1 s'_1$ then there exists s'_2 with $s_2 \Rightarrow_2 s'_2$ such that $\langle s'_1, s'_2 \rangle \in \mathcal{B}$.
 - c) for each s'_2 , if $s_2 \Rightarrow_2 s'_2$, then there exists s'_1 with $s_1 \Rightarrow_1 s'_1$, such that $\langle s'_1, s'_2 \rangle \in \mathcal{B}$.
- or
2. s_1 and s_2 are both filter application states.
 - a) $s_1 \simeq_{\text{cx}} s_2$
 - b) for each s'_1 , if $s_1 \Rightarrow_1 s'_1$ then there exists t_1, \dots, t_n (for $n \geq 0$) and s'_2 with

$$s_2 \Rightarrow_2 t_1 \Rightarrow_2 \dots \Rightarrow_2 t_n \Rightarrow_2 s'_2$$

such that $\langle s'_1, s'_2 \rangle \in \mathcal{B}$, t_1, \dots, t_n are filter application intermediate states, s'_1 and s'_2 are stable states.

- c) for each s'_2 , if

$$s_2 \Rightarrow_2 t_1 \Rightarrow_2 \dots \Rightarrow_2 t_n \Rightarrow_2 s'_2$$

(for $n \geq 0$) with t_1, \dots, t_n are filter application intermediate states and s'_2 is a stable state, then there exists s'_1 with $s_1 \Rightarrow_1 s'_1$, such that $\langle s'_1, s'_2 \rangle \in \mathcal{B}$, s'_1 and s'_2 are stable states.

■

Let $\mathcal{T}_1 = \langle \Delta, T_{\text{cx}}, \Sigma_1, s_{01}, \text{abox}_1, \text{ctx}, \text{actsrc}, \text{actpar}, \text{fa}, \text{fd}, \Rightarrow_1 \rangle$ be a fine-grained transition system, and $\mathcal{T}_2 = \langle \Delta, T, \Sigma_2, s_{02}, \text{abox}_2, \Rightarrow_2 \rangle$ be a KB transition system, a state $s_1 \in \Sigma_1$ is *AJ-bisimilar* to $s_2 \in \Sigma_2$, written $s_1 \sim_{\text{AJ}} s_2$, if there exists an AJ-bisimulation relation \mathcal{B} between \mathcal{T}_1 and \mathcal{T}_2 such that $\langle s_1, s_2 \rangle \in \mathcal{B}$. A transition system \mathcal{T}_1 is *AJ-bisimilar* to \mathcal{T}_2 , written $\mathcal{T}_1 \sim_{\text{AJ}} \mathcal{T}_2$, if there exists an AJ-bisimulation relation \mathcal{B} between \mathcal{T}_1 and \mathcal{T}_2 such that $\langle s_{01}, s_{02} \rangle \in \mathcal{B}$.

In the following two lemmas, we show some important properties of AJ-bisimilar states and transition systems that will be useful later to show that we can recast the verification of B-AGKABs into S-GKABs. Essentially, we show that given a fine-grained transition system \mathcal{T}_1 and a typed KB transition system \mathcal{T}_2 such that they are AJ-bisimilar, we have that \mathcal{T}_1 satisfies a $\mu\mathcal{L}_{\text{CTX}}^{\text{Alt}}$ formula implies that \mathcal{T}_2 satisfies the same formula modulo translation t_j^A and vice versa.

Lemma 8.31. *Let $\mathcal{T}_1 = \langle \Delta, T_{cx}, \Sigma_1, s_{01}, abox_1, ctx, actsrc, actpar, fa, fd, \Rightarrow_1 \rangle$ be a fine-grained transition system, and $\mathcal{T}_2 = \langle \Delta, T, \Sigma_2, s_{02}, abox_2, \Rightarrow_2 \rangle$ be a typed KB transition system. Consider two states $s_1 \in \Sigma_1$ and $s_2 \in \Sigma_2$ such that $s_1 \sim_{\text{AJ}} s_2$. Then for every formula Φ (in NNF) of $\mu\mathcal{L}_{\text{CTX}}^{\text{Alt}}$, and every valuations v_1 and v_2 that assign to each of its free variables a constant $c_1 \in \text{ADOM}(abox_1(s_1))$ and $c_2 \in \text{ADOM}(abox_2(s_2))$, such that $c_1 = c_2$, we have that*

$$\mathcal{T}_1, s_1 \models \Phi v_1 \text{ if and only if } \mathcal{T}_2, s_2 \models t_j^A(\Phi) v_2.$$

Proof. Similar to the combination of the proof for Lemmas 4.32 and 6.46 by also considering that by the definition of typed KB transition system and fine-grained transition system, both of them start from a stable state. Additionally, for the transition among the states, it is always be the case that there are three intermediate states (i.e., service call evaluation state, context change state, and filter application state) between the two stable states. On the other hand, the $\mu\mathcal{L}_{\text{CTX}}^{\text{Alt}}$ formula always have four consecutive modal operators. Hence, it is easy to see that we always verify query and context expression in the given $\mu\mathcal{L}_{\text{CTX}}^{\text{Alt}}$ formula over a stable state. \square

Lemma 8.32. *Consider a fine-grained transition system \mathcal{T}_1 , and a typed KB transition system \mathcal{T}_2 such that $\mathcal{T}_1 \sim_{\text{AJ}} \mathcal{T}_2$. For every closed $\mu\mathcal{L}_{\text{CTX}}^{\text{Alt}}$ formula Φ (in NNF), we have:*

$$\mathcal{T}_1 \models \Phi \text{ if and only if } \mathcal{T}_2 \models t_j^A(\Phi)$$

Proof. Let $\mathcal{T}_1 = \langle \Delta, T_{cx}, \Sigma_1, s_{01}, abox_1, ctx, actsrc, actpar, fa, fd, \Rightarrow_1 \rangle$, and $\mathcal{T}_2 = \langle \Delta, T, \Sigma_2, s_{02}, abox_2, \Rightarrow_2 \rangle$. By the definition of AJ-bisimilar transition system, we have that $s_{01} \sim_{\text{AJ}} s_{02}$. Thus, we obtain the proof as a consequence of Lemma 8.31, due to the fact that

$$\mathcal{T}_1, s_{01} \models \Phi \text{ if and only if } \mathcal{T}_2, s_{02} \models t_j^A(\Phi)$$

\square

8.2.1.3 Reducing the Verification of B-AGKABs into S-GKABs

In this section we show that we can recast the verification of B-AGKABs into S-GKABs. In the following two lemmas we aim to show that the transition system of a B-AGKAB \mathcal{G}_A is AJ-bisimilar to the transition system of the corresponding S-GKAB $\tau_B^A(\mathcal{G}_A)$ that is obtained via translation τ_B^A .

Lemma 8.33. *Let \mathcal{G}_A be a B-AGKAB with transition system $\Upsilon_{\mathcal{G}_A}^{f_{\text{B}}^{\text{cx}}}$, and let $\tau_B^A(\mathcal{G}_A)$ be its corresponding S-GKAB (with transition system $\Upsilon_{\tau_B^A(\mathcal{G}_A)}^{f_{\text{B}}^{\text{S}}}$) obtain through τ_B^A .*

Consider a state $s_{cx} = \langle id_{cx}, A_{cx}, m_{cx}, C, \delta_{cx} \rangle$ of $\Upsilon_{\mathcal{G}_A}^{f_{\text{B}}^{\text{cx}}}$ and a state $s_s = \langle A_s, m_s, \delta_s \rangle$ of $\Upsilon_{\tau_B^A(\mathcal{G}_A)}^{f_{\text{B}}^{\text{S}}}$. If the following hold:

1. s_{cx} and s_s are having the same state type,
2. $s_{cx} \simeq_{cx} s_s$ (see Definition 8.29),
3. $m_{cx} = m_s$,
4. $fa(s_{cx}) = \text{ADD}(A_s)$ (see Definition 8.11 for the definition of $\text{ADD}(A_s)$),
5. $fd(s_{cx}) = \text{DEL}(A_s)$ (see Definition 8.12 for the definition of $\text{DEL}(A_s)$),
6. $\delta_s = \kappa_B^A(\delta_{cx})$ (if s_{cx} and s_s are stable states),
7. $\text{Actsrc}(\alpha) \in A_s$ (if s_{cx} and s_s are not stable states, and $\text{actsrc}(s_{cx}) = \alpha$),
8. $\{\text{Par}_1(\sigma(p_1)), \dots, \text{Par}_m(\sigma(p_m))\} = \text{PAR}(A_s)$ (if s_{cx} and s_s are not stable states, and $\text{actpar}(s_{cx}) = \sigma$),

then $s_{cx} \sim_{AJ} s_s$,

Proof. Let

- $\mathcal{G}_A = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$, and
 $\mathcal{R}_{\mathcal{G}_A}^{fcx} = \langle \Delta, T_{cx}, \Sigma_1, s_{01}, \text{abox}_1, \text{ctx}, \text{actsrc}, \text{actpar}, fa, fd, \Rightarrow_1 \rangle$,
- $\tau_B^A(\mathcal{G}_A) = \langle T_D, A_0 \cup A_{C_0}, \Gamma', \delta' \rangle$, and
 $\mathcal{R}_{\tau_B^A(\mathcal{G}_A)}^{fs} = \langle \Delta, T_D, \Sigma_2, s_{02}, \text{abox}_2, \Rightarrow_2 \rangle$.

First, observe that by Lemma 8.27, we have that $\mathcal{R}_{\tau_B^A(\mathcal{G}_A)}^{fs}$ is typed KB transition system. Hence, in $\mathcal{R}_{\tau_B^A(\mathcal{G}_A)}^{fs}$ we have the alternation between state types.

To prove the lemma, in the following we show that for every state $s'_{cx} = \langle id'_{cx}, A'_{cx}, m'_{cx}, C', \delta'_{cx} \rangle$ such that $s_{cx} \Rightarrow_1 s'_{cx}$, there exists $s'_s = \langle A'_s, m'_s, \delta'_s \rangle$ such that $s_s \Rightarrow_2 s'_s$ and the following hold

1. s'_{cx} and s'_s are having the same state type,
2. $s'_{cx} \simeq_{cx} s'_s$,
3. $m'_{cx} = m'_s$,
4. $fa(s'_{cx}) = \text{ADD}(A'_s)$,
5. $fd(s'_{cx}) = \text{DEL}(A'_s)$,
6. $\delta'_s = \kappa_B^A(\delta'_{cx})$ (if s'_{cx} and s'_s are stable states),
7. $\text{Actsrc}(\alpha) \in A'_s$ (if s'_{cx} and s'_s are not stable states, and $\text{actsrc}(s'_{cx}) = \alpha$),
8. $\{\text{Par}_1(\sigma(p_1)), \dots, \text{Par}_m(\sigma(p_m))\} = \text{PAR}(A'_s)$ (if s'_{cx} and s'_s are not stable states, and $\text{actpar}(s'_{cx}) = \sigma$),

To show the claim, we have to separately discuss the case in which:

1. both s_{cx} and s_s are stable states,
2. both s_{cx} and s_s are service call evaluation states,
3. both s_{cx} and s_s are context change states, and
4. both s_{cx} and s_s are filter application states.

Base case. trivially true from the shape of the initial states of $\mathcal{R}_{\mathcal{G}_A}^{fcx}$ and $\mathcal{R}_{\tau_B^A(\mathcal{G}_A)}^{fs}$.

Case 1 - s_{cx} and s_s are stable states: Now we have to show that for every state $s'_{cx} = \langle id'_{cx}, A'_{cx}, m'_{cx}, C', \delta'_{cx} \rangle$ such that $s_{cx} \Rightarrow_1 s'_{cx}$, there exists $s'_s = \langle A'_s, m'_s, \delta'_s \rangle$ such that $s_s \Rightarrow_2 s'_s$ and the following hold

- (i) s'_{cx} and s'_s are service call evaluation states,
- (ii) $s'_{cx} \simeq_{cx} s'_s$,
- (iii) $m'_{cx} = m'_s$,
- (iv) $fa(s'_{cx}) = \text{ADD}(A'_s)$,
- (v) $fd(s'_{cx}) = \text{DEL}(A'_s)$,

- (vi) $actsrc(s'_{cx}) = \alpha$ and $Actsrc(\alpha) \in A'_s$,
- (vii) $actpar(s'_{cx}) = \sigma$ and $\{Par_1(\sigma(p_1)), \dots, Par_m(\sigma(p_m))\} = \text{PAR}(A'_s)$.

Now, by definition of $\mathcal{R}_{\mathcal{G}_A}^{f_{B}^{cx}}$, since $s_{cx} \Rightarrow_1 s'_{cx}$, we have

$$\langle A_{cx}, m_{cx}, C, \delta_{cx} \rangle \xrightarrow{\alpha\sigma, f_B^{cx}} \langle A_{cx}, m_{cx}, C, \delta'_{cx} \rangle.$$

Hence, by the definition of $\xrightarrow{\alpha\sigma, f_B^{cx}}$ (Definition 8.1), we have that:

- σ is a legal parameter assignment for α in A_{cx} w.r.t. context C and an action invocation **pick** $\langle Q(\vec{p}), \varphi_C \rangle. \alpha(\vec{p})$ (i.e., $\text{ASK}(Q\sigma, T_{cx}^C, A_{cx}) = \text{true}$). Notice that w.l.o.g. **pick** $\langle Q(\vec{p}), \varphi_C \rangle. \alpha(\vec{p})$ is the next instruction that should be executed in δ_{cx} .
- $C \cup \Phi_{\mathbb{D}} \models \varphi_C$.

Additionally, we have $actsrc(s'_{cx}) = \alpha$, and $actpar(s'_{cx}) = \sigma$.

Now, on the other hand, since $\delta_s = \kappa_B^A(\delta_{cx})$, by the definition of κ_B^A (see Definition 8.25), we have

$$\begin{aligned} \kappa_B^A(\mathbf{pick} \langle Q(\vec{p}), \varphi_C \rangle. \alpha(\vec{p})) &= \mathbf{pick} Q'(\vec{p}).\alpha_a(\vec{p}); \mathbf{pick} \text{true}.\alpha_b(); \delta_{\Pi_C}; \\ &\quad \mathbf{pick} \text{true}.\alpha^u(); \delta_b^{T_{cx}}; \mathbf{pick} \text{true}.\alpha_{temp}^-() \end{aligned}$$

where **pick** $Q'(\vec{p}).\alpha_a(\vec{p}); \mathbf{pick} \text{true}.\alpha_b()$ is a split action invocation obtained from **pick** $\langle Q(\vec{p}), \varphi_C \rangle. \alpha(\vec{p})$ as in Definition 8.22. Thus, we have that $Q' = Q_{cx} \wedge q_{\varphi_C}$. Since $C \cup \Phi_{\mathbb{D}} \models \varphi_C$ and q_{φ_C} only use context dimension concept, by Lemma 6.33, it is easy to see that $\text{CERT}(q_{\varphi_C}, T_{\mathbb{D}}, A_s) = \text{true}$. Furthermore, by Lemma 6.35, we have that $\text{CERT}(Q, T_{cx}^C, A_{cx}) = \text{CERT}(Q_{cx}, T_{\mathbb{D}}, A_s)$. Therefore, now we can construct σ_s that maps parameters of α' to constants in $\text{ADOM}(A_s)$ such that $\sigma_c = \sigma_s$. Therefore, by definition of split action (see Definition 8.21), we have $Actsrc(\alpha) \in A'_s$, and $\{Par_1(\sigma(p_1)), \dots, Par_m(\sigma(p_m))\} = \text{PAR}(A'_s)$. Next, by the definition of split action it can be seen easily that

- $m'_{cx} = m'_s$, because α_a does not involve any service call and α does not update the service call map).
- $fa(s'_{cx}) = \text{ADD}(A'_s)$, because $fa(s'_{cx})$ is the set of ABox assertions in $\text{ADD}(T_{cx}^C, A_{cx}, \alpha\sigma)$ (excluding the ground skolem terms) and α_a only add information about ABox assertions to be added in which their construction do not involve any service call.
- $fd(s'_{cx}) = \text{DEL}(A'_s)$, because $fd(s'_{cx}) = \text{DEL}(T_{cx}^C, A_{cx}, \alpha\sigma)$ and α_a add information about all ABox assertions to be deleted by action α .
- $s'_{cx} \simeq_{cx} s'_s$, because the context does not change and also the ABox stay the same. Additionally, α_a only add some assertions that is made by special reserved concept/role names.
- s'_{cx} and s'_s are service call evaluation states. Because α_a adds the assertion $\text{St}(\text{servCl})$ and also by the construction of $\mathcal{R}_{\mathcal{G}_A}^{f_{B}^{cx}}$.

Since s'_{cx} and s'_s are service call evaluation states, then the case 2 is applicable.

Case 2 - s_{cx} and s_s are service call evaluation states. Now we have to show that for every state $s'_{cx} = \langle id'_{cx}, A'_{cx}, m'_{cx}, C', \delta'_{cx} \rangle$ such that $s_{cx} \Rightarrow_1 s'_{cx}$, there exists $s'_s = \langle A'_s, m'_s, \delta'_s \rangle$ such that $s_s \Rightarrow_2 s'_s$ and the following hold

- (i) s'_{cx} and s'_s are context change states,
- (ii) $s'_{cx} \simeq_{cx} s'_s$,
- (iii) $m'_{cx} = m'_s$,
- (iv) $fa(s'_{cx}) = \text{ADD}(A'_s)$,
- (v) $fd(s'_{cx}) = \text{DEL}(A'_s)$,
- (vi) $actsrc(s'_{cx}) = \alpha$ and $Actsrc(\alpha) \in A'_s$,
- (vii) $actpar(s'_{cx}) = \sigma$ and $\{Par_1(\sigma(p_1)), \dots, Par_m(\sigma(p_m))\} = \text{PAR}(A'_s)$.

Now, let $actsrc(s_{cx}) = \alpha$ with parameters p_1, \dots, p_m and $actpar(s_{cx}) = \sigma$. By definition of $\Upsilon_{\mathcal{G}_A}^{f_{B}^{cx}}$, since $s_{cx} \Rightarrow_1 s'_{cx}$, we have that there exists $\theta \in \text{EVAL}(\text{ADD}(T_{cx}^C, A, \alpha\sigma))$, and $m'_{cx} = m_{cx} \cup \theta$.

On the other hand, by the definition of κ_B^A (see Definition 8.25) and the states alternation in the typed KB transition system, the next action invocation to be executed in the state s_s is **pick true**. $\alpha_b()$ where α_b is the second split action of α . Thus, by the definition of execution semantics of S-GKABs, the transition $s_s \Rightarrow_2 s'_s$ involves the following:

- the execution of action α_b with legal parameter assignments σ_s where σ_s is an empty substitution because α_b is a 0-ary action.
- the service call substitution θ_s which evaluates all ground skolem terms generated by the execution of α_b .

By the definition of split action, It is easy to see that $\theta_s = \theta$ because $m_{cx} = m_s$ and α_b is the second split action of α which add all of assertions that is added by α and involve service call (α_b and α involve the same service call).

Next, by the definition of split action, it can be seen easily that:

- $m'_{cx} = m'_s$ because $\theta_s = \theta$, $m_{cx} = m_s$, $m'_{cx} = m_{cx} \cup \theta$ and $m'_s = m_s \cup \theta_s$.
- $fa(s'_{cx}) = \text{ADD}(A'_s)$, because
 - (1) $fa(s_{cx}) = \text{ADD}(A_s)$ and they contain the set of ABox assertions in $\text{ADD}(T_{cx}^C, A_{cx}, \alpha\sigma)$ (excluding the ground skolem terms),
 - (2) $fa(s'_{cx}) = \text{ADD}(T_{cx}^C, A_{cx}, \alpha\sigma)\theta$,
 - (3) The set of ground skolem terms in $\text{ADD}(T_{cx}^C, A_{cx}, \alpha\sigma)$ and $\text{ADD}(T_D, A_s, \alpha_b\sigma_s)$ are the same because of the definition of split action and also the following:
 - * $\{Par_1(\sigma(p_1)), \dots, Par_m(\sigma(p_m))\} = \text{PAR}(A_s)$,
 - * α_b is the second split action of α .
 - (4) α_b is the second split action of α which add all of assertions that is added by α and involve service call (α_b and α involve the same service call). Let A_{α_b} be the set of ABox assertions that are made by added fact marker concept/role names and added by α_b , then it is easy to see that $\text{ADD}(A'_s) = \text{ADD}(A_s) \cup A_{\alpha_b}$.
 - (5) Now, since we also have that α and α_b involve the same service call, $m_{cx} = m_s$ and $\theta = \theta_s$, it is easy to see that $\text{ADD}(A'_s) = \text{ADD}(T_{cx}^C, A_{cx}, \alpha\sigma)\theta$.
- $fd(s'_{cx}) = \text{DEL}(A'_s)$, because $fd(s_{cx}) = \text{DEL}(A_s) = \text{DEL}(T_{cx}^C, A_{cx}, \alpha\sigma)$, $fd(s'_{cx}) = fd(s_{cx})$, and α_b does not add any ABox assertion made by the deleted fact marker concept/role names.

- $s'_{cx} \simeq_{cx} s'_s$, because $s_{cx} \simeq_{cx} s_s$, the context does not change and also the ABox (apart from the special markers) stay the same. Additionally, α_b only add/delete some assertions that is made by special reserved concept/role names.
- $Actsrc(\alpha) \in A'_s$ simply because $Actsrc(\alpha) \in A_s$ and α_b does nothing w.r.t. the concept assertion made by $Actsrc$,
- $\{Par_1(\sigma(p_1)), \dots, Par_m(\sigma(p_m))\} = PAR(A'_s)$ because $\{Par_1(\sigma(p_1)), \dots, Par_m(\sigma(p_m))\} = PAR(A_s)$, and α_b does nothing w.r.t. the concept assertion made by action parameter concept names.
- s'_{cx} and s'_s are context change states. Because of the construction of $\mathcal{R}_{\mathcal{G}_A}^{f_{B}^{cx}}$ as well as because α_b removes the assertion $St(servCl)$ and adds the assertion $St(ctxChg)$.

Since s'_{cx} and s'_s are context change states, then the case 3 is applicable.

Case 3 - s_{cx} and s_s are context change states. Now we have to show that for every state $s'_{cx} = \langle id'_{cx}, A'_{cx}, m'_{cx}, C', \delta'_{cx} \rangle$ such that $s_{cx} \Rightarrow_1 s'_{cx}$, there exists $s'_s = \langle A'_s, m'_s, \delta'_s \rangle$ such that $s_s \Rightarrow_2 s'_s$ and the following hold

- (i) s'_{cx} and s'_s are filter application states,
- (ii) $s'_{cx} \simeq_{cx} s'_s$,
- (iii) $m'_{cx} = m'_s$,
- (iv) $fa(s'_{cx}) = ADD(A'_s)$,
- (v) $fd(s'_{cx}) = DEL(A'_s)$,
- (vi) $actsrc(s'_{cx}) = \alpha$ and $Actsrc(\alpha) \in A'_s$,
- (vii) $actpar(s'_{cx}) = \sigma$ and $\{Par_1(\sigma(p_1)), \dots, Par_m(\sigma(p_m))\} = PAR(A'_s)$.

Now, let $actsrc(s_{cx}) = \alpha$ with parameters p_1, \dots, p_m and $actpar(s_{cx}) = \sigma$. By definition of $\mathcal{R}_{\mathcal{G}_A}^{f_{B}^{cx}}$, since $s_{cx} \Rightarrow_1 s'_{cx}$, we have that $\langle A_{cx}, C, C' \rangle \in \text{CTX-CHG}$. Therefore, by Definition 6.16, there exists a context-evolution rule $\langle Q, \varphi_C \rangle \mapsto C_{new}$ in Π_C s.t.:

1. $ASK(Q, T_{cx}^C, A_{cx})$ is true;
2. $C \cup \Phi_D \models \varphi_C$;
3. for every context dimension $d \in \mathbb{D}$ s.t. $[d \rightsquigarrow v] \in C_{new}$, we have $[d \rightsquigarrow v] \in C'$;
4. for every context dimension $d \in \mathbb{D}$ s.t. $[d \rightsquigarrow v] \in C$, and there does not exist any v_2 s.t. $[d \rightsquigarrow v_2] \in C_{new}$, we have $[d \rightsquigarrow v] \in C'$.

On the other hand, by the definition of κ_B^A (see Definition 8.25) and the states alternation in the typed KB transition system, the part of the program to be executed in the state s_s is δ_{Π_C} . By the definition of δ_{Π_C} , there exist an action invocation **pick** $Q'.\alpha_C^s()$ that is obtained from $\langle Q, \varphi_C \rangle \mapsto C_{new}$. Since $s_{cx} \simeq_{cx} s_s$ it is easy to see that there exists A'_s such that $\text{CTX}(A'_s) = A_{C'}$ and A'_s is obtained by the execution of α_C^s .

Next, it can be seen easily that:

- $m'_{cx} = m'_s$ because $m_{cx} = m_s$ and both transitions $s_{cx} \Rightarrow_1 s'_{cx}$ and $s_s \Rightarrow_2 s'_s$ do not involve any service calls.

- $fa(s'_{cx}) = \text{ADD}(A'_s)$, because $fa(s_{cx}) = \text{ADD}(A_s)$, $fa(s'_{cx}) = fa(s_{cx})$, and $\text{ADD}(A'_s) = \text{ADD}(A_s)$ because α_C^s does not add any ABox assertion made by the added fact marker concept/role names.
- $fd(s'_{cx}) = \text{DEL}(A'_s)$, because $fd(s_{cx}) = \text{DEL}(A_s)$, $fd(s'_{cx}) = fd(s_{cx})$, and $\text{DEL}(A'_s) = \text{DEL}(A_s)$ because α_C^s does not add any ABox assertion made by the deleted fact marker concept/role names.
- $s'_{cx} \simeq_{cx} s'_s$, because $\text{CTX}(A'_s) = A_{C'}$ and because α_C^s only changes the context related information.
- $\text{Actsrc}(\alpha) \in A'_s$ simply because $\text{Actsrc}(\alpha) \in A_s$ and α_C^s does nothing w.r.t. the concept assertion made by Actsrc ,
- $\{Par_1(\sigma(p_1)), \dots, Par_m(\sigma(p_m))\} = \text{PAR}(A'_s)$ because $\{Par_1(\sigma(p_1)), \dots, Par_m(\sigma(p_m))\} = \text{PAR}(A_s)$, and α_C^s does nothing w.r.t. the concept assertion made by action parameter concept names.
- s'_{cx} and s'_s are filter application states. Because α_C^s removes the assertion $\text{St}(\text{ctxChg})$ and adds the assertion $\text{St}(\text{filt})$ and also because of the construction of $\Upsilon_{\mathcal{G}_A}^{f_B^{cx}}$.

Since s'_{cx} and s'_s are filter application states, then the case 4 is applicable.

Case 4 - s_{cx} and s_s are filter application states. Now we have to show that for every state $s'_{cx} = \langle id'_{cx}, A'_{cx}, m'_{cx}, C', \delta'_{cx} \rangle$ such that $s_{cx} \Rightarrow_1 s'_{cx}$, there exists t_1, \dots, t_n (for $n \geq 0$), and $s'_s = \langle A'_s, m'_s, \delta'_s \rangle$ such that $s_s \Rightarrow_2 t_1 \Rightarrow_2 \dots \Rightarrow_2 t_n \Rightarrow_2 s'_s$ and the following hold

- (i) s'_{cx} and s'_s are stable states,
- (ii) $s'_{cx} \simeq_{cx} s'_s$,
- (iii) $m'_{cx} = m'_s$,
- (iv) $fa(s'_{cx}) = \text{ADD}(A'_s)$,
- (v) $fd(s'_{cx}) = \text{DEL}(A'_s)$,
- (vi) $\delta'_s = \kappa_B^A(\delta'_{cx})$.

By the definition of $\Upsilon_{\mathcal{G}_A}^{f_B^{cx}}$, since $s_{cx} \Rightarrow_1 s'_{cx}$, we have that

$$\langle A_{cx}, fa(s_{cx}), fd(s_{cx}), C', A'_{cx} \rangle \in f_B^{cx},$$

On the other hand, by the definition of κ_B^A (see Definition 8.25) and the states alternation in the typed KB transition system, the part of the program to be executed in the state s_s is

$$\text{pick true.}\alpha^u(); \delta_b^{T_{cx}}; \text{pick true.}\alpha_{temp}^-().$$

Hence, by considering those program above that need to be executed and also the correctness of b-repair program $\delta_b^{T_{cx}}$ in simulating b-repair computation as in Section 7.2.1.2, we can easily see that there exists states t_1, \dots, t_n (for $n \geq 0$), and $s'_s = \langle A'_s, m'_s, \delta'_s \rangle$ with $s_s \Rightarrow_2 t_1 \Rightarrow_2 \dots \Rightarrow_2 t_n \Rightarrow_2 s'_s$ and the following hold:

- t_1, \dots, t_n are filter application intermediate states, because α^u removes the assertion $\text{St}(\text{filt})$ and add the assertion $\text{St}(\text{int})$.
- s'_{cx} and s'_s are stable states, because α_{temp}^- removes the assertion $\text{St}(\text{int})$ and also by the construction of $\Upsilon_{\mathcal{G}_A}^{f_B^{cx}}$.
- $s'_{cx} \simeq_{cx} s'_s$, because of the following:

- * $s_{cx} \simeq_{cx} s_s$,
- * the transition $s_s \Rightarrow_2 t_1 \Rightarrow_2 \dots \Rightarrow_2 t_n \Rightarrow_2 s'_s$ do not change any context information,
- * the correctness of b-repair program $\delta_b^{T_{cx}}$ in simulating b-repair computation, and essentially A'_s is the b-repair of A_s .
- $m'_{cx} = m'_s$, because both of the transitions $s_{cx} \Rightarrow_1 s'_{cx}$ and $s_s \Rightarrow_2 t_1 \Rightarrow_2 \dots \Rightarrow_2 t_n \Rightarrow_2 s'_s$ do not involve any service calls.
- $fa(s'_{cx}) = \text{ADD}(A'_s)$, because α^u removes all ABox assertions made by added fact marker concept/role names.
- $fd(s'_{cx}) = \text{DEL}(A'_s)$, because α^u removes all ABox assertions made by deleted fact marker concept/role names.
- $\delta'_s = \kappa_B^A(\delta'_{cx})$, by the definition of κ_B^A (see Definition 8.25) and also the construction of $\Upsilon_{\mathcal{G}_A}^{f_{cx}^B}$.

The other direction can be shown similarly. \square

Lemma 8.34. *Given a B-AGKAB \mathcal{G}_A with transition system $\Upsilon_{\mathcal{G}_A}^{f_{cx}^B}$, let $\tau_B^A(\mathcal{G}_A)$ be its corresponding S-GKAB (with transition system $\Upsilon_{\tau_B^A(\mathcal{G}_A)}^{f_S}$) obtained through τ_B^A . We have that $\Upsilon_{\mathcal{G}_A}^{f_{cx}^B} \sim_{AJ} \Upsilon_{\tau_B^A(\mathcal{G}_A)}^{f_S}$*

Proof. Let

- $\mathcal{G}_A = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$, and $\Upsilon_{\mathcal{G}_A}^{f_{cx}^B} = \langle \Delta, T_{cx}, \Sigma_1, s_{01}, abox_1, ctx, actsrc, actpar, fa, fd, \Rightarrow_1 \rangle$,
- $\tau_B^A(\mathcal{G}_A) = \langle T_D, A'_0, \Gamma', \delta' \rangle$, and $\Upsilon_{\tau_B^A(\mathcal{G}_A)}^{f_S} = \langle \Delta, T_D, \Sigma_2, s_{02}, abox_2, \Rightarrow_2 \rangle$.

We have that $s_{01} = \langle id, A_0, m_{cx}, C_0, \delta \rangle$ and $s_{02} = \langle A'_0, m_s, \delta' \rangle$ where $m_{cx} = m_s = \emptyset$. By the definition of κ_B^A and τ_B^A , we also have: (i) $s_{01} \simeq_{cx} s_{02}$, (ii) s_{01} and s_{02} are stable states, (iii) $fa(s_{01}) = \text{ADD}(s_{02}) = \emptyset$ (iv) $fd(s_{01}) = \text{DEL}(s_{02}) = \emptyset$ (v) $\delta' = \kappa_B^{cx}(\delta)$. Hence, by Lemma 8.33, we have $s_{01} \sim_{AJ} s_{02}$. Therefore, by the definition of AJ-bisimulation, we have $\Upsilon_{\mathcal{G}_A}^{f_{cx}^B} \sim_{AJ} \Upsilon_{\tau_B^A(\mathcal{G}_A)}^{f_S}$. \square

Next, we show that the verification of $\mu\mathcal{L}_{CTX}^{Alt}$ properties over B-AGKABs can be recast as verification of $\mu\mathcal{L}_A^{EQL}$ over S-GKABs as follows.

Theorem 8.35. *Given a B-AGKAB \mathcal{G}_A and a closed $\mu\mathcal{L}_{CTX}^{Alt}$ property Φ (in NNF), we have*

$$\Upsilon_{\mathcal{G}_A}^{f_{cx}^B} \models \Phi \text{ if and only if } \Upsilon_{\tau_B^A(\mathcal{G}_A)}^{f_S} \models t_j^A(\Phi)$$

Proof. By Lemma 8.34, we have that $\Upsilon_{\mathcal{G}_A}^{f_{cx}^B} \sim_{AJ} \Upsilon_{\tau_B^A(\mathcal{G}_A)}^{f_S}$. Hence, by Lemma 8.32, we have that for every $\mu\mathcal{L}_{CTX}^{Alt}$ property Φ

$$\Upsilon_{\mathcal{G}_A}^{f_{cx}^B} \models \Phi \text{ if and only if } \Upsilon_{\tau_B^A(\mathcal{G}_A)}^{f_S} \models t_j^A(\Phi)$$

\square

8.2.2 Verification of C-AGKABs

This section is dedicated to show the reduction of $\mu\mathcal{L}_{\text{CTX}}^{\text{Alt}}$ verification over C-AGKABs into the $\mu\mathcal{L}_A^{\text{EQL}}$ verification over S-GKABs. Basically, we start by presenting how we translate C-AGKABs into S-GKABs, and also how we translate $\mu\mathcal{L}_{\text{CTX}}^{\text{Alt}}$ formulas into $\mu\mathcal{L}_A^{\text{EQL}}$ w.r.t. our purpose. Then, here we introduce a specific notion of bisimulation that will be used to show that we can recast the verification of C-AGKABs into S-GKABs.

8.2.2.1 Translating C-AGKABs to S-GKABs

To the aim of translating C-AGKABs into S-GKABs, in the following we first introduce the notion of program translation for the program in C-AGKABs.

Program Translation
 κ_C^A

Definition 8.36 (Program Translation κ_C^A). Given a C-AGKABs $\mathcal{G}_A = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$, we define a *translation* κ_C^A which translates a program δ into a program δ' inductively as follows:

$$\begin{aligned} \kappa_C^A(\text{pick } \langle Q(\vec{p}), \varphi_C \rangle . \alpha(\vec{p})) &= \text{pick } Q'(\vec{p}) . \alpha_a(\vec{p}); \text{pick true} . \alpha_b(); \\ &\quad \delta_{\Pi_C}; \text{pick true} . \alpha^u(); \text{pick true} . \alpha_c^{T_{cx}}() \\ \kappa_C^A(\varepsilon) &= \varepsilon \\ \kappa_C^A(\delta_1 | \delta_2) &= \kappa_C^A(\delta_1) | \kappa_C^A(\delta_2) \\ \kappa_C^A(\delta_1; \delta_2) &= \kappa_C^A(\delta_1); \kappa_C^A(\delta_2) \\ \kappa_C^A(\text{if } \varphi \text{ then } \delta_1 \text{ else } \delta_2) &= \text{if } \varphi \text{ then } \kappa_C^A(\delta_1) \text{ else } \kappa_C^A(\delta_2) \\ \kappa_C^A(\text{while } \varphi \text{ do } \delta) &= \text{while } \varphi \text{ do } \kappa_C^A(\delta) \end{aligned}$$

where

- $\text{pick } Q'(\vec{p}) . \alpha_a(\vec{p}); \text{pick true} . \alpha_b(\vec{p})$ is a split action invocation obtained from $\text{pick } \langle Q(\vec{p}), \varphi_C \rangle . \alpha(\vec{p})$ as in Definition 8.22,
- δ_{Π_C} is a context-change program obtained from Π_C as in Definition 6.39 except that it is formed by sole action invocation obtained from context evolution rule as in Definition 8.23.
- α^u is an update action (see Definition 8.24).
- $\alpha_c^{T_{cx}}$ is a context-sensitive c-repair action over T_{cx} as in Definition 7.24, except that we remove the effect $\text{true} \rightsquigarrow \text{del } \{\text{State}(\text{temp})\}$ from $\text{EFF}(\alpha_c^{T_{cx}})$ and add the effect $\text{true} \rightsquigarrow \text{del } \{\text{St}(\text{int})\}$ into $\text{EFF}(\alpha_c^{T_{cx}})$. ■

We now step further to define the translation from C-AGKABs into S-GKABs as follows by also utilizing the program translation κ_C^A defined above.

Translation from
C-AGKAB to
S-GKAB

Definition 8.37 (Translation from C-AGKAB to S-GKAB). We define a translation τ_C^A that, given a C-AGKAB $\mathcal{G}_A = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$, produces an S-GKAB $\tau_C^A(\mathcal{G}_A) = \langle T_{\mathbb{D}}, A_0 \cup A_{C_0}, \Gamma', \delta' \rangle$, where

- $T_{\mathbb{D}}$ is a TBox obtained from a set of context dimensions \mathbb{D} (see Definition 6.29),
- A_{C_0} is an ABox obtained from C_0 (see Definition 6.31),
- $\Gamma' = \Gamma_\alpha \cup \Gamma_C \cup \{\alpha^u, \alpha_c^{T_{cx}}\}$ where:

- Γ_α is obtained from Γ such that for each action $\alpha \in \Gamma$, we have $\alpha_1, \alpha_2 \in \Gamma_\alpha$ where α_1 and α_2 are split action obtained from α (see Definition 8.21),
- Γ_C is obtained from Π_C such that for each context-evolution rule $\langle Q, \varphi_C \rangle \mapsto C_{new}$ in Π_C , we have $\alpha_C \in \Gamma_C$ where α_C is a sole action obtained from the context-evolution rule $\langle Q, \varphi_C \rangle \mapsto C_{new}$ (see Definition 8.23),
- α^u is an update action (see Definition 8.24).
- $\alpha_c^{T_{cx}}$ is a context-sensitive c-repair action over T_{cx} as in Definition 7.24, except that we remove the effect $\text{true} \rightsquigarrow \mathbf{del} \{\text{State}(temp)\}$ from $\text{EFF}(\alpha_c^{T_{cx}})$ and add the effect $\text{true} \rightsquigarrow \mathbf{del} \{\text{St}(int)\}$ into $\text{EFF}(\alpha_c^{T_{cx}})$.
- $\delta' = \kappa_C^A(\delta)$.

■

The $\mu\mathcal{L}_{\text{CTX}}^{\text{Alt}}$ property Φ over a C-AGKAB \mathcal{G}_A can then be recast as a corresponding property over an S-GKAB $\tau_C^A(\mathcal{G}_{cx})$ using the following formula translation:

Definition 8.38 (Translation t_s^A). We define a translation t_s^A that transforms an arbitrary $\mu\mathcal{L}_{\text{CTX}}^{\text{Alt}}$ formula Φ into another $\mu\mathcal{L}_A^{\text{EQL}}$ formula Φ' inductively by recurring over the structure of Φ as follows:

Translation t_s^A

$$\begin{aligned}
\bullet t_s^A(Q) &= Q_{cx} \\
\bullet t_s^A(\varphi_C) &= q_{\varphi_C} \\
\bullet t_s^A(\neg\Phi) &= \neg t_s^A(\Phi) \\
\bullet t_s^A(\exists x.\Phi) &= \exists x.t_s^A(\Phi) \\
\bullet t_s^A(\Phi_1 \vee \Phi_2) &= t_s^A(\Phi_1) \vee t_s^A(\Phi_2) \\
\bullet t_s^A(\mu Z.\Phi) &= \mu Z.t_s^A(\Phi) \\
\bullet t_s^A(\odot \odot \odot \langle \neg \rangle \Phi) &= \odot \odot \odot \langle \neg \rangle t_s^A(\Phi) \\
\bullet t_s^A(\odot \odot \odot [\neg] \Phi) &= \odot \odot \odot [\neg] t_s^A(\Phi)
\end{aligned}$$

where \odot is either box $[\neg]$ or diamond $\langle \neg \rangle$ modal operator.

■

8.2.2.2 Alternating Skip-one Bisimulation (AS-Bisimulation)

Here we introduce the notion of alternating skip-one bisimulation (AS-bisimulation) and show an important lemma about the situation where two AS-bisimilar transition systems can not be distinguished by certain temporal properties. This bisimulation relation is an important tool to show that we can recast the verification of $\mu\mathcal{L}_{\text{CTX}}^{\text{Alt}}$ over C-AGKABS (resp. E-AGKABS) into the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ over S-GKABS.

Definition 8.39 (Alternating Skip-one Bisimulation (AS-Bisimulation)).

Alternating Skip-one
Bisimulation
(AS-Bisimulation)

Let $\mathcal{T}_1 = \langle \Delta, T_{cx}, \Sigma_1, s_{01}, abox_1, ctx, actsrc, actpar, fa, fd, \Rightarrow_1 \rangle$ be a fine-grained transition system, and $\mathcal{T}_2 = \langle \Delta, T, \Sigma_2, s_{02}, abox_2, \Rightarrow_2 \rangle$ be a typed KB transition system, with $\text{ADOM}(abox_1(s_{01})) \subseteq \Delta$, $\text{ADOM}(abox_2(s_{02})) \subseteq \Delta$, and s_{01} as well as s_{02} are stable states. An alternating skip-one bisimulation (AS-Bisimulation) between \mathcal{T}_1 and \mathcal{T}_2 is a relation $\mathcal{B} \subseteq \Sigma_1 \times \Sigma_2$ such that $\langle s_1, s_2 \rangle \in \mathcal{B}$ implies that one of the following condition hold:

1. s_1 as well as s_2 are either stable states, service call evaluation states, or context change states and we have the following:

- a) $s_1 \simeq_{cx} s_2$
 - b) for each s'_1 , if $s_1 \Rightarrow_1 s'_1$ then there exists s'_2 with $s_2 \Rightarrow_2 s'_2$ such that $\langle s'_1, s'_2 \rangle \in \mathcal{B}$, s'_1 and s'_2 are service call evaluation states.
 - c) for each s'_2 , if $s_2 \Rightarrow_2 s'_2$, then there exists s'_1 with $s_1 \Rightarrow_1 s'_1$, such that $\langle s'_1, s'_2 \rangle \in \mathcal{B}$, s'_1 and s'_2 are context change states.
2. s_1 and s_2 are filter application states.
- a) $s_1 \simeq_{cx} s_2$
 - b) for each s'_1 , if $s_1 \Rightarrow_1 s'_1$ then there exists t and s'_2 with

$$s_2 \Rightarrow_2 t \Rightarrow_2 s'_2$$

such that $\langle s'_1, s'_2 \rangle \in \mathcal{B}$, t is a filter application intermediate state, s'_1 and s'_2 are stable states.

- c) for each s'_2 , if

$$s_2 \Rightarrow_2 t \Rightarrow_2 s'_2,$$

with t is a filter application intermediate state, then there exists s'_1 with $s_1 \Rightarrow_1 s'_1$, such that $\langle s'_1, s'_2 \rangle \in \mathcal{B}$, s'_1 and s'_2 are stable states. ■

Let $\mathcal{T}_1 = \langle \Delta, T_{cx}, \Sigma_1, s_{01}, abox_1, ctx, actsrc, actpar, fa, fd, \Rightarrow_1 \rangle$ be a fine-grained transition system, and $\mathcal{T}_2 = \langle \Delta, T, \Sigma_2, s_{02}, abox_2, \Rightarrow_2 \rangle$ be a KB transition system, a state $s_1 \in \Sigma_1$ is *AS-bisimilar* to $s_2 \in \Sigma_2$, written $s_1 \sim_{AS} s_2$, if there exists an AS-bisimulation relation \mathcal{B} between \mathcal{T}_1 and \mathcal{T}_2 such that $\langle s_1, s_2 \rangle \in \mathcal{B}$. A transition system \mathcal{T}_1 is *AS-bisimilar* to \mathcal{T}_2 , written $\mathcal{T}_1 \sim_{AS} \mathcal{T}_2$, if there exists an AS-bisimulation relation \mathcal{B} between \mathcal{T}_1 and \mathcal{T}_2 such that $\langle s_{01}, s_{02} \rangle \in \mathcal{B}$.

In the following two lemmas we show some important properties of AS-bisimilar states and transition systems that will be useful later to show that we can recast the verification of C-AGKABs (as well as E-AGKABs) into S-GKABs.

Lemma 8.40. *Let $\mathcal{T}_1 = \langle \Delta, T_{cx}, \Sigma_1, s_{01}, abox_1, ctx, actsrc, actpar, fa, fd, \Rightarrow_1 \rangle$ be a fine-grained transition system, and $\mathcal{T}_2 = \langle \Delta, T, \Sigma_2, s_{02}, abox_2, \Rightarrow_2 \rangle$ be a typed KB transition system, Consider two states $s_1 \in \Sigma_1$ and $s_2 \in \Sigma_2$ such that $s_1 \sim_{AS} s_2$. Then for every formula Φ of $\mu\mathcal{L}_{CTX}^{Alt}$, and every valuations v_1 and v_2 that assign to each of its free variables a constant $c_1 \in \text{ADOM}(abox_1(s_1))$ and $c_2 \in \text{ADOM}(abox_2(s_2))$, such that $c_1 = c_2$, we have that*

$$\mathcal{T}_1, s_1 \models \Phi v_1 \text{ if and only if } \mathcal{T}_2, s_2 \models t_s^A(\Phi) v_2.$$

Proof. Similar to the combination of the proof for Lemmas 5.41 and 6.46 by also considering that by the definition of typed KB transition system and fine-grained transition system, both of them start from a stable state. Additionally, for the transition among the states in \mathcal{T}_1 , it is always be the case that there are three intermediate states (i.e., service call evaluation state, context change state, and filter application state) between the two stable states. Moreover, for the case of \mathcal{T}_2 , we have that there is an additional states namely a filter application intermediate state. On the other hand, the $\mu\mathcal{L}_{CTX}^{Alt}$ formula always have four consecutive modal operators and the translation t_s^A always add an additional modal operator for the translation of modal operators in $\mu\mathcal{L}_{CTX}^{Alt}$. Hence, it is easy to see that we always verify query and context expression in the given $\mu\mathcal{L}_{CTX}^{Alt}$ formula over a stable state. □

Lemma 8.41. *Consider a fine-grained transition system Υ_1 , and a typed KB transition system Υ_2 such that $\Upsilon_1 \sim_{\text{AS}} \Upsilon_2$. For every closed $\mu\mathcal{L}_{\text{CTX}}^{\text{Alt}}$ formula Φ , we have:*

$$\Upsilon_1 \models \Phi \text{ if and only if } \Upsilon_2 \models t_s^A(\Phi)$$

Proof. Let $\Upsilon_1 = \langle \Delta, T_{cx}, \Sigma_1, s_{01}, abox_1, ctx, actsrc, actpar, fa, fd, \Rightarrow_1 \rangle$, and $\Upsilon_2 = \langle \Delta, T, \Sigma_2, s_{02}, abox_2, \Rightarrow_2 \rangle$. By the definition of AS-bisimilar transition system, we have that $s_{01} \sim_{\text{AS}} s_{02}$. Thus, we obtain the proof as a consequence of Lemma 8.40, due to the fact that

$$\Upsilon_1, s_{01} \models \Phi \text{ if and only if } \Upsilon_2, s_{02} \models t_s^A(\Phi)$$

□

8.2.2.3 Reducing the Verification of C-AGKABs into S-GKABs

We now proceed to show that we can recast the verification of C-AGKABs into S-GKABs. In the following two lemmas we aim to show that the transition system of a C-AGKAB \mathcal{G}_A is AS-bisimilar to the transition system of the corresponding S-GKAB $\tau_C^A(\mathcal{G}_A)$ that is obtained via translation τ_C^A .

Lemma 8.42. *Let \mathcal{G}_A be a C-AGKAB with transition system $\Upsilon_{\mathcal{G}_A}^{f_{cx}}$, and let $\tau_C^A(\mathcal{G}_A)$ be its corresponding S-GKAB (with transition system $\Upsilon_{\tau_C^A(\mathcal{G}_A)}^{f_s}$) obtain through τ_C^A .*

Consider a state $s_{cx} = \langle id_{cx}, A_{cx}, m_{cx}, C, \delta_{cx} \rangle$ of $\Upsilon_{\mathcal{G}_A}^{f_{cx}}$ and a state $s_s = \langle A_s, m_s, \delta_s \rangle$ of $\Upsilon_{\tau_C^A(\mathcal{G}_A)}^{f_s}$. If the following hold:

1. s_{cx} and s_s are having the same state type,
2. $s_{cx} \simeq_{cx} s_s$ (see Definition 8.29),
3. $m_{cx} = m_s$,
4. $fa(s_{cx}) = \text{ADD}(A_s)$ (see Definition 8.11 for the definition of $\text{ADD}(A_s)$),
5. $fd(s_{cx}) = \text{DEL}(A_s)$ (see Definition 8.12 for the definition of $\text{DEL}(A_s)$),
6. $\delta_s = \kappa_C^A(\delta_{cx})$ (if s_{cx} and s_s are stable states),
7. $\text{Acts}rc(\alpha) \in A_s$ (if s_{cx} and s_s are not stable states, and $\text{acts}rc(s_{cx}) = \alpha$),
8. $\{\text{Par}_1(\sigma(p_1)), \dots, \text{Par}_m(\sigma(p_m))\} = \text{PAR}(A_s)$ (if s_{cx} and s_s are not stable states, and $\text{actpar}(s_{cx}) = \sigma$),

then $s_{cx} \sim_{\text{AS}} s_s$,

Proof. Similar to the proof of Lemma 8.33. The different is only in the case when s_{cx} and s_s are both filter application states. Here, instead of applying the b-repair program, we apply the c-repair action. Similar to Theorem 5.51, we can also easily show the correctness of context-sensitive c-repair action. The important observation is that the context-sensitive c-repair action do the repair based on the context, i.e., it only consider those assertion in the TBox that “holds” under the corresponding context. □

Lemma 8.43. *Given a C-AGKAB \mathcal{G}_A with transition system $\Upsilon_{\mathcal{G}_A}^{f_{cx}}$, let $\tau_C^A(\mathcal{G}_A)$ be its corresponding S-GKAB (with transition system $\Upsilon_{\tau_C^A(\mathcal{G}_A)}^{f_s}$) obtained through τ_C^A . We have that $\Upsilon_{\mathcal{G}_A}^{f_{cx}} \sim_{\text{AJ}} \Upsilon_{\tau_C^A(\mathcal{G}_A)}^{f_s}$*

Proof. Let

- $\mathcal{G}_A = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$, and
 $\Upsilon_{\mathcal{G}_A}^{fcx} = \langle \Delta, T_{cx}, \Sigma_1, s_{01}, abox_1, ctx, actsrc, actpar, fa, fd, \Rightarrow_1 \rangle$,
- $\tau_C^A(\mathcal{G}_A) = \langle T_D, A'_0, \Gamma', \delta' \rangle$, and
 $\Upsilon_{\tau_C^A(\mathcal{G}_A)}^{fs} = \langle \Delta, T_D, \Sigma_2, s_{02}, abox_2, \Rightarrow_2 \rangle$.

We have that $s_{01} = \langle id, A_0, m_{cx}, C_0, \delta \rangle$ and $s_{02} = \langle A'_0, m_s, \delta' \rangle$ where $m_{cx} = m_s = \emptyset$. By the definition of κ_C^A and τ_C^A , we also have: (i) $s_{01} \simeq_{cx} s_{02}$, (ii) s_{01} and s_{02} are stable states, (iii) $fa(s_{01}) = \text{ADD}(s_{02}) = \emptyset$ (iv) $fd(s_{01}) = \text{DEL}(s_{02}) = \emptyset$ (v) $\delta' = \kappa_C^A(\delta)$. Hence, by Lemma 8.42, we have $s_{01} \sim_{AS} s_{02}$. Therefore, by the definition of AS-bisimulation, we have $\Upsilon_{\mathcal{G}_A}^{fcx} \sim_{AS} \Upsilon_{\tau_C^A(\mathcal{G}_A)}^{fs}$. \square

Now, we show that the verification of $\mu\mathcal{L}_{CTX}^{Alt}$ properties over C-AGKABs can be recast as verification of $\mu\mathcal{L}_A^{EQL}$ over S-GKABs as follows.

Theorem 8.44. *Given a C-AGKAB \mathcal{G}_A and a closed $\mu\mathcal{L}_{CTX}^{Alt}$ property Φ , we have*

$$\Upsilon_{\mathcal{G}_A}^{fcx} \models \Phi \text{ if and only if } \Upsilon_{\tau_C^A(\mathcal{G}_A)}^{fs} \models t_s^A(\Phi)$$

Proof. By Lemma 8.43, we have that $\Upsilon_{\mathcal{G}_A}^{fcx} \sim_{AS} \Upsilon_{\tau_C^A(\mathcal{G}_A)}^{fs}$. Hence, by Lemma 8.41, we have that for every $\mu\mathcal{L}_{CTX}^{Alt}$ property Φ

$$\Upsilon_{\mathcal{G}_A}^{fcx} \models \Phi \text{ if and only if } \Upsilon_{\tau_C^A(\mathcal{G}_A)}^{fs} \models t_s^A(\Phi)$$

\square

8.2.3 Verification of E-AGKABs

This section is devoted to show that we can recast the verification of $\mu\mathcal{L}_{CTX}^{Alt}$ over E-AGKABs into the verification of $\mu\mathcal{L}_A^{EQL}$ over S-GKABs. We open this section by explaining how we translate E-AGKABs into S-GKABs, and also how we transform $\mu\mathcal{L}_{CTX}^{Alt}$ formulas into $\mu\mathcal{L}_A^{EQL}$ w.r.t. our purpose. By making use the AS-bisimulation defined before, later we show that we can reduce the verification of $\mu\mathcal{L}_{CTX}^{Alt}$ over E-AGKABs into the verification of $\mu\mathcal{L}_A^{EQL}$ over S-GKABs.

8.2.3.1 Translating E-AGKABs to S-GKABs

In order to define our generic translation which transforms any E-AGKABs into S-GKABs, we first introduce the program translation for the program inside E-AGKABs as follows.

Program Translation
 κ_E^A

Definition 8.45 (Program Translation κ_E^A). Given an E-AGKABs $\mathcal{G}_A = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$, we define a translation κ_E^A which translates a program δ into a program δ' inductively as follows:

$$\begin{aligned}
\kappa_E^A(\mathbf{pick} \langle Q(\vec{p}), \varphi_C \rangle . \alpha(\vec{p})) &= \mathbf{pick} Q'(\vec{p}) . \alpha_a(\vec{p}); \mathbf{pick} \mathbf{true} . \alpha_b(); \\
&\quad \delta_{\Pi_C}; \mathbf{pick} \mathbf{true} . \alpha^u(); \mathbf{pick} \neg Q_{\text{unsatECQ}}^{T_a} . \alpha_e^{T_{cx}}() \\
\kappa_E^A(\varepsilon) &= \varepsilon \\
\kappa_E^A(\delta_1 | \delta_2) &= \kappa_E^A(\delta_1) | \kappa_E^A(\delta_2) \\
\kappa_E^A(\delta_1; \delta_2) &= \kappa_E^A(\delta_1); \kappa_E^A(\delta_2) \\
\kappa_E^A(\mathbf{if} \varphi \mathbf{then} \delta_1 \mathbf{else} \delta_2) &= \mathbf{if} \varphi \mathbf{then} \kappa_E^A(\delta_1) \mathbf{else} \kappa_E^A(\delta_2) \\
\kappa_E^A(\mathbf{while} \varphi \mathbf{do} \delta) &= \mathbf{while} \varphi \mathbf{do} \kappa_E^A(\delta)
\end{aligned}$$

where

- $\mathbf{pick} Q'(\vec{p}) . \alpha_a(\vec{p}); \mathbf{pick} \mathbf{true} . \alpha_b(\vec{p})$ is a split action invocation obtained from $\mathbf{pick} \langle Q(\vec{p}), \varphi_C \rangle . \alpha(\vec{p})$ as in Definition 8.22,
- δ_{Π_C} is a context-change program obtained from Π_C as in Definition 6.39 except that it is formed by sole action invocation obtained from context evolution rule as in Definition 8.23.
- α^u is an update action as in Definition 8.24 except that we replace the effect $N^a(x) \rightsquigarrow \mathbf{add} \{N(x)\}, \mathbf{del} \{N^a(x)\}$ in $\text{EFF}(\alpha^u)$ with $N^a(x) \rightsquigarrow \mathbf{add} \{N(x)\}$. The different is only that we do not delete the assertions made by added fact marker concept/role names.
- $Q_{\text{unsatECQ}}^{T_a}$ is a context-sensitive Q-UNSAT-ECQ over T_a (see Definition 7.11), where T_a is obtained from T_{cx} by renaming each concept name N in T_{cx} into N^a (similarly for roles). Thus, with this mechanism, we can block any further execution when the newly added assertions are inconsistent.
- $\alpha_e^{T_{cx}}$ is a context-sensitive evolution action over T_{cx} as in Definition 7.31, except that we replace the effect $\mathbf{true} \rightsquigarrow \mathbf{del} \{\text{State}(\text{temp})\}$ in $\text{EFF}(\alpha_e^{T_{cx}})$ with the effect $\mathbf{true} \rightsquigarrow \mathbf{del} \{\text{St}(\text{int})\}$.

■

By utilizing the program translation κ_E^A defined above, we define the translation from E-AGKABs into S-GKABs as follows:

Definition 8.46 (Translation from E-AGKAB to S-GKAB). We define a translation τ_E^A that, given an E-AGKAB $\mathcal{G}_A = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$, produces an S-GKAB $\tau_C^A(\mathcal{G}_A) = \langle T_D, A_0 \cup A_{C_0}, \Gamma', \delta' \rangle$, where

*Translation from
E-AGKAB to
S-GKAB*

- T_D is a TBox obtained from a set of context dimensions ID (see Definition 6.29),
- A_{C_0} is an ABox obtained from C_0 (see Definition 6.31),
- $\Gamma' = \Gamma_\alpha \cup \Gamma_C \cup \{\alpha^u, \alpha_e^{T_{cx}}\}$ where:
 - Γ_α is obtained from Γ such that for each action $\alpha \in \Gamma$, we have $\alpha_1, \alpha_2 \in \Gamma_\alpha$ where α_1 and α_2 are split action obtained from α (see Definition 8.21),
 - Γ_C is obtained from Π_C such that for each context-evolution rule $\langle Q, \varphi_C \rangle \mapsto C_{new}$ in Π_C , we have $\alpha_C \in \Gamma_C$ where α_C is a sole action obtained from the context-evolution rule $\langle Q, \varphi_C \rangle \mapsto C_{new}$ (see Definition 8.23),
 - α^u is an update action (see Definition 8.24).
 - $\alpha_e^{T_{cx}}$ is a context-sensitive evolution action over T_{cx} as in Definition 7.31, except that we replace the effect $\mathbf{true} \rightsquigarrow \mathbf{del} \{\text{State}(\text{temp})\}$ in $\text{EFF}(\alpha_e^{T_{cx}})$ with the effect $\mathbf{true} \rightsquigarrow \mathbf{del} \{\text{St}(\text{int})\}$.

- $\delta' = \kappa_E^A(\delta)$.

■

8.2.3.2 Reducing the Verification of E-AGKABs into S-GKABs

We now step forward to show that the verification of E-AGKABs can be reduced into the verification S-GKABs. In the following two lemmas we aim to show that the transition system of an E-AGKABs \mathcal{G}_A is AS-bisimilar to the transition system of the corresponding S-GKAB $\tau_E^A(\mathcal{G}_A)$ (that is obtained via translation τ_E^A).

Lemma 8.47. *Let \mathcal{G}_A be an E-AGKAB with transition system $\Upsilon_{\mathcal{G}_A}^{f_{cx}}$, and let $\tau_E^A(\mathcal{G}_A)$ be its corresponding S-GKAB (with transition system $\Upsilon_{\tau_E^A(\mathcal{G}_A)}^{f_s}$) obtain through τ_E^A . Consider a state $s_{cx} = \langle id_{cx}, A_{cx}, m_{cx}, C, \delta_{cx} \rangle$ of $\Upsilon_{\mathcal{G}_A}^{f_{cx}}$ and a state $s_s = \langle A_s, m_s, \delta_s \rangle$ of $\Upsilon_{\tau_E^A(\mathcal{G}_A)}^{f_s}$. If the following hold:*

1. s_{cx} and s_s are having the same state type,
2. $s_{cx} \simeq_{cx} s_s$ (see Definition 8.29),
3. $m_{cx} = m_s$,
4. $fa(s_{cx}) = \text{ADD}(A_s)$ (see Definition 8.11 for the definition of $\text{ADD}(A_s)$),
5. $fd(s_{cx}) = \text{DEL}(A_s)$ (see Definition 8.12 for the definition of $\text{DEL}(A_s)$),
6. $\delta_s = \kappa_E^A(\delta_{cx})$ (if s_{cx} and s_s are stable states),
7. $\text{Actsrc}(\alpha) \in A_s$ (if s_{cx} and s_s are not stable states, and $\text{actsrc}(s_{cx}) = \alpha$),
8. $\{\text{Par}_1(\sigma(p_1)), \dots, \text{Par}_m(\sigma(p_m))\} = \text{PAR}(A_s)$ (if s_{cx} and s_s are not stable states, and $\text{actpar}(s_{cx}) = \sigma$),

then $s_{cx} \sim_{\text{AS}} s_s$,

Proof. Similar to the proof of Lemma 8.33. The different is only in the case when s_{cx} and s_s are both filter application states, instead of applying the b-repair program, we apply the evolution action. Another aspect to observe in order to complete the proof is as follows:

- Similar to Lemmas 5.58 and 5.59, we can also easily show the correctness of context-sensitive evolution action that it performs the bold-evolution computation. The important observation is that the context-sensitive evolution action performs the bold-evolution based on the context, i.e., it only consider those assertion in the TBox that “holds” under the corresponding context.
- As it can be seen from the translation κ_E^A (see Definition 8.45), before executing the evolution action, we also check the consistency of the updates w.r.t. the TBox under the new context. This guarantees that we fulfill the requirement in Definition 7.7 that the updates must consistent w.r.t. the TBox under the new context.

□

Lemma 8.48. *Given an E-AGKAB \mathcal{G}_A with transition system $\Upsilon_{\mathcal{G}_A}^{f_{cx}}$, let $\tau_E^A(\mathcal{G}_A)$ be its corresponding S-GKAB (with transition system $\Upsilon_{\tau_E^A(\mathcal{G}_A)}^{f_s}$) obtained through τ_E^A . We have that $\Upsilon_{\mathcal{G}_A}^{f_{cx}} \sim_{\text{AS}} \Upsilon_{\tau_E^A(\mathcal{G}_A)}^{f_s}$*

Proof. Let

- $\mathcal{G}_A = \langle T_{cx}, A_0, \Gamma, \delta, C_0, \Pi_C \rangle$, and
 $\Upsilon_{\mathcal{G}_A}^{f_{cx}} = \langle \Delta, T_{cx}, \Sigma_1, s_{01}, abox_1, ctx, actsrc, actpar, fa, fd, \Rightarrow_1 \rangle$,
- $\tau_E^A(\mathcal{G}_A) = \langle T_D, A'_0, \Gamma', \delta' \rangle$, and
 $\Upsilon_{\tau_E^A(\mathcal{G}_A)}^{f_S} = \langle \Delta, T_D, \Sigma_2, s_{02}, abox_2, \Rightarrow_2 \rangle$.

We have that $s_{01} = \langle id, A_0, m_{cx}, C_0, \delta \rangle$ and $s_{02} = \langle A'_0, m_s, \delta' \rangle$ where $m_{cx} = m_s = \emptyset$. By the definition of κ_E^A and τ_E^A , we also have: (i) $s_{01} \simeq_{cx} s_{02}$, (ii) s_{01} and s_{02} are stable states, (iii) $fa(s_{01}) = \text{ADD}(s_{02}) = \emptyset$ (iv) $fd(s_{01}) = \text{DEL}(s_{02}) = \emptyset$ (v) $\delta' = \kappa_E^A(\delta)$. Hence, by Lemma 8.47, we have $s_{01} \sim_{AS} s_{02}$. Therefore, by the definition of AS-bisimulation, we have $\Upsilon_{\mathcal{G}_A}^{f_{cx}} \sim_{AS} \Upsilon_{\tau_E^A(\mathcal{G}_A)}^{f_S}$. \square

Having Lemma 8.48 in hand, we can now easily show that the verification of $\mu\mathcal{L}_{CTX}^{Alt}$ properties over E-AGKAB can be recast as verification of $\mu\mathcal{L}_A^{EQL}$ over S-GKAB as follows.

Theorem 8.49. *Given an E-AGKAB \mathcal{G}_A and a closed $\mu\mathcal{L}_{CTX}^{Alt}$ property Φ , we have*

$$\Upsilon_{\mathcal{G}_A}^{f_{cx}} \models \Phi \text{ if and only if } \Upsilon_{\tau_E^A(\mathcal{G}_A)}^{f_S} \models t_s^A(\Phi)$$

Proof. By Lemma 8.48, we have that $\Upsilon_{\mathcal{G}_A}^{f_{cx}} \sim_{AS} \Upsilon_{\tau_E^A(\mathcal{G}_A)}^{f_S}$. Hence, by Lemma 8.41, we have that for every $\mu\mathcal{L}_{CTX}^{Alt}$ property Φ

$$\Upsilon_{\mathcal{G}_A}^{f_{cx}} \models \Phi \text{ if and only if } \Upsilon_{\tau_E^A(\mathcal{G}_A)}^{f_S} \models t_s^A(\Phi)$$

\square

8.2.4 Putting it all together: Verification of AGKABS

Putting together all results above, from Theorems 4.54, 8.35, 8.44 and 8.49 we get that verification of B-AGKABS, C-AGKABS, and E-AGKABS can be compiled into verification of KABs, by first translating them into S-GKABS, and then into KABs.

Theorem 8.50. *Verification of $\mu\mathcal{L}_{CTX}^{Alt}$ properties over B-AGKABS, C-AGKABS, and E-AGKABS can be reduced to verification over KABs.*

Proof. The proof can be easily obtained since from Theorems 8.35, 8.44 and 8.49, we can reduce the verification of $\mu\mathcal{L}_{CTX}^{Alt}$ over B-AGKABS, C-AGKABS, and E-AGKABS as verification of $\mu\mathcal{L}_A^{EQL}$ over S-GKABS and then by Theorem 4.54 we can recast the verification of $\mu\mathcal{L}_A^{EQL}$ over S-GKABS as verification of $\mu\mathcal{L}_A^{EQL}$ over KABs. Thus, combining all of those ingredients, we have that the claim is proven. \square

8.2.5 Verification of Run-bounded AGKABS

Even more interesting, our reductions from B-AGKABS, C-AGKABS, and E-AGKABS into S-GKABS preserve run-boundedness.

Lemma 8.51. *Let \mathcal{G}_A be a B-AGKABS and $\tau_B^A(\mathcal{G}_A)$ be its corresponding S-GKAB. We have \mathcal{G}_A is run-bounded if and only if $\tau_B^A(\mathcal{G}_A)$ is run-bounded.*

Proof. Let

- $\mathcal{R}_{\mathcal{G}_A}^{f_{cx}^B}$ be the transition system of \mathcal{G}_A , and
- $\mathcal{R}_{\tau_B^A(\mathcal{G}_A)}^{f_S}$ be the transition system of $\tau_B^A(\mathcal{G}_A)$.

The claim can be easily shown by observing the following:

- the translation τ_B^A essentially do the following:
 1. Splits each action α into two actions α_a and α_b where the former do the computation of α that do not involve any service calls while the latter do the computation of α that involve service calls.
 2. Appends the split actions with a context-change program that simulates context evolution.
 3. Appends context-change program with an additional program to simulate the b-repair computation.
- the split actions α_a and α_b of α essentially only split the computation that is done by α into two steps.
- the program that is used to simulate the context evolution does not inject unbounded number of new constants. In fact, we only reserve a constant \mathbf{c} to simulate the context (i.e., to construct the ABox assertions that represent the context dimension assignments).
- the program/actions that simulate the b-repair computation never inject new additional constants, but only remove facts causing inconsistency,
- by Lemma 8.34, we have that $\mathcal{R}_{\mathcal{G}_A}^{f_{cx}^B} \sim_{AJ} \tau_B^A(\mathcal{G}_A)$. Thus, basically they are “equivalent” modulo filter application intermediate states (states containing $\text{St}(int)$) and also by considering that they represent context information in a different way.

□

Lemma 8.52. *Let \mathcal{G}_A be a C-AGKAB and $\tau_C^A(\mathcal{G}_A)$ be its corresponding S-GKAB. We have \mathcal{G}_A is run-bounded if and only if $\tau_C^A(\mathcal{G}_A)$ is run-bounded.*

Proof. Similar to the proof of Lemma 8.51. □

Lemma 8.53. *Let \mathcal{G}_A be a E-AGKAB and $\tau_E^A(\mathcal{G}_A)$ be its corresponding S-GKAB. We have \mathcal{G}_A is run-bounded if and only if $\tau_E^A(\mathcal{G}_A)$ is run-bounded.*

Proof. Similar to the proof of Lemma 8.51. □

To close our tour on this chapter, we show the result on the verification of $\mu\mathcal{L}_{\text{CTX}}^{\text{Alt}}$ properties over run-bounded B-AGKABs, C-AGKABs, and E-AGKABs as follows.

Theorem 8.54. *Verification of $\mu\mathcal{L}_{\text{CTX}}^{\text{Alt}}$ properties over run-bounded B-AGKABs, C-AGKABs, and E-AGKABs are decidable, and reducible to standard μ -calculus finite-state model checking.*

Proof. By Lemmas 8.51 to 8.53, the translation from B-AGKABs, C-AGKABs, and E-AGKABs to S-GKABs preserves run-boundedness. Thus, the claim follows by combining Theorems 8.35, 8.44 and 8.49 and Theorem 4.56. □

8.3 Emulating Standard GKABS in Alternating GKABS

We have seen so far that we can recast the verification of B-AGKABS, C-AGKABS, and E-AGKABS into S-GKABS. Now, we show that we can also do the other direction. In particular, we show that we can recast the verification of S-GKABS into the verification of B-AGKABS. The reductions from C-AGKABS and E-AGKABS into S-GKABS can be done similarly.

8.3.1 Transforming S-GKABS into B-AGKABS

Similar to Sections 5.4, 6.6 and 7.3, some challenges in order to reduce B-AGKABS into S-GKABS is to prevent the repair and also the context change. Therefore, to cope with this situation, here we simply adopt our previous approach. Another important observation in order to reduce the verification of S-GKABS into B-AGKABS is that essentially S-GKABS ignore the various quantification among all sources of non-determinisms. Thus, to mimics this situation, we simply need to translate the given $\mu\mathcal{L}_A^{\text{EQL}}$ formulas over S-GKABS into the corresponding $\mu\mathcal{L}_{\text{CTX}}^{\text{Alt}}$ formulas over B-AGKABS by duplicating the modal operator eight times (I.e., we translate $\langle - \rangle \Phi$ into $\langle - \rangle \langle - \rangle \langle - \rangle \langle - \rangle \langle - \rangle \langle - \rangle \langle - \rangle \langle - \rangle \Phi$ and similarly for $[-]$). The reason why we need to duplicates the modal operator eight times is that because a single action execution in S-GKABS will correspond to two action execution in B-AGKABS (One for the same action execution, and one for the action that does the inconsistency check). Additionally, notice that each action execution in B-AGKABS requires four transitions until it reaches another stable state, we then need to quadruplicate the modal operator for each action execution.

In the following we fix a set \mathbb{D} of context dimension containing only a single context dimension d (i.e., $\mathbb{D} = \{d\}$). Moreover, $d \in \mathbb{D}$ has a tree shaped finite value domain $\langle \text{Dom}(d), \prec_d \rangle$ where $\text{Dom}(d)$ contains only a single value \top_d (i.e., $\text{Dom}(d) = \top_d$).

To translate the program in the given S-GKABS, we basically can reuse the program translation κ_{sic} in Definition 7.43. We then define the following translation that transform S-GKABS into B-AGKABS as follows.

Definition 8.55 (Translation from S-GKAB to B-AGKAB). We define a translation τ_{gba} that, given an S-GKAB $\mathcal{G} = \langle T, A_0, \Gamma, \delta \rangle$, produces a B-AGKAB $\tau_{\text{gba}}(\mathcal{G}) = \langle T_{\text{gba}}, A_0, \Gamma', \delta', C_0, \Pi_C \rangle$, where

*Translation from
S-GKAB to
B-AGKAB*

- T_{gba} is obtained from T such that for each positive inclusion assertion $t \in T$, we have $\langle t : \varphi \rangle$ where $\varphi = [d \rightsquigarrow \top_d]$,
- $\Gamma' = \Gamma_\alpha \cup \{\alpha_\perp\}$ where
 - Γ_α is obtained from Γ such that for each $\alpha \in \Gamma$, we have $\alpha' \in \Gamma_\alpha$ where $\text{EFF}(\alpha') = \text{EFF}(\alpha) \cup \{\text{true} \rightsquigarrow \text{add } \{\text{State}(\text{temp})\}\}$,
 - α_\perp is a 0-ary action of the form $\alpha_\perp() : \{\text{true} \rightsquigarrow \text{del } \{\text{State}(\text{temp})\}\}$.
- $\delta' = \kappa_{\text{sic}}(\delta)$.
- $C_0 = \{[d \rightsquigarrow \top_d]\}$,
- $\Pi_C = \{\langle \text{true}, [d \rightsquigarrow \top_d] \rangle \mapsto \{[d \rightsquigarrow \top_d]\}\}$

■

The $\mu\mathcal{L}_A^{\text{EQL}}$ property Φ over an S-GKAB \mathcal{G} can then be recast as a corresponding property over a B-AGKAB $\tau_{\text{gba}}(\mathcal{G})$ using the following formula translation:

$\mu\mathcal{L}_A^{\text{EQL}}$ Formula
Translation t_{sba}

Definition 8.56 (Translation t_{sba}). We define a *translation* t_{sba} that takes a $\mu\mathcal{L}_A^{\text{EQL}}$ formula Φ as an input and produces a $\mu\mathcal{L}_{\text{CTX}}^{\text{Alt}}$ formula $t_{sba}(\Phi)$ by recurring over the structure of Φ as follows:

- $t_{sba}(Q) = Q$
- $t_{sba}(\neg\Phi) = \neg t_{sba}(\Phi)$
- $t_{sba}(\exists x.\Phi) = \exists x.t_{sba}(\Phi)$
- $t_{sba}(\Phi_1 \vee \Phi_2) = t_{sba}(\Phi_1) \vee t_{sba}(\Phi_2)$
- $t_{sba}(\mu Z.\Phi) = \mu Z.t_{sba}(\Phi)$
- $t_{sba}(\langle \rightarrow \rangle \Phi) = \langle \rightarrow \rangle \langle \rightarrow \rangle \langle \rightarrow \rangle \langle \rightarrow \rangle \langle \rightarrow \rangle \langle \rightarrow \rangle \langle \rightarrow \rangle \langle \rightarrow \rangle t_{sba}(\Phi)$

■

Having the translations τ_{sba} and t_{sba} in hand, we show it later that $\gamma_{\mathcal{G}}^{fs} \models \Phi$ if and only if $\gamma_{\tau_{sba}(\mathcal{G})}^{f_{\mathcal{B}}^{cx}} \models t_{sba}(\Phi)$ which consequently means that the verification of $\mu\mathcal{L}_A^{\text{EQL}}$ over S-GKABs can be reduced to the corresponding verification of $\mu\mathcal{L}_{\text{CTX}}^{\text{Alt}}$ over B-AGKABs. Towards this aim, in the next section we introduce a special bisimulation relation that will ease to reduce the verification of S-GKABs into B-AGKABs.

8.3.2 Skip-Seven Bisimulation (S7-Bisimulation)

In this section we the notion of S7-Bisimulation and show that two S7-bisimilar transition system can not be distinguished by $\mu\mathcal{L}_A^{\text{EQL}}$ formula modulo the translation t_{sba} . Then, we will see in the next section that the transition system of an S-GKAB is actually S7-bisimilar to the transition system of its corresponding B-AGKABs that is obtained via τ_{sba} .

Skip-Seven
Bisimulation
(S7-Bisimulation)

Definition 8.57 (Skip-Seven Bisimulation (S7-Bisimulation)).

Let $\mathcal{T}_1 = \langle \Delta, T, \Sigma_1, s_{01}, abox_1, \Rightarrow_1 \rangle$ be a KB transition system, and $\mathcal{T}_2 = \langle \Delta, T_{cx}, \Sigma_2, s_{02}, abox_2, ctx, \Rightarrow_2 \rangle$ be context-sensitive transition system, with $\text{ADOM}(abox_1(s_{01})) \subseteq \Delta$ and $\text{ADOM}(abox_2(s_{02})) \subseteq \Delta$. A *skip-seven bisimulation* (S7-Bisimulation) between \mathcal{T}_1 and \mathcal{T}_2 is a relation $\mathcal{B} \subseteq \Sigma_1 \times \Sigma_2$ such that $\langle s_1, s_2 \rangle \in \mathcal{B}$ implies that:

1. $abox_1(s_1) = abox_2(s_2)$
2. for each s'_1 , if $s_1 \Rightarrow_1 s'_1$ then there exists t_i (for $i \in \{1, \dots, 7\}$) and s'_2 with

$$s_2 \Rightarrow_2 t_1 \Rightarrow_2 \dots \Rightarrow_2 t_7 \Rightarrow_2 s'_2$$

such that $\langle s'_1, s'_2 \rangle \in \mathcal{B}$, $\text{State}(temp) \notin abox_2(s'_2)$ and $\text{State}(temp) \in abox_2(t_i)$ for $i \in \{1, \dots, 7\}$.

3. for each s'_2 , if

$$s_2 \Rightarrow_2 t_1 \Rightarrow_2 \dots \Rightarrow_2 t_7 \Rightarrow_2 s'_2$$

with $\text{State}(temp) \in abox_2(t_i)$ for $i \in \{1, \dots, 7\}$ and $\text{State}(temp) \notin abox_2(s'_2)$, then there exists s'_1 with $s_1 \Rightarrow_1 s'_1$, such that $\langle s'_1, s'_2 \rangle \in \mathcal{B}$.

■

Let $\mathcal{T}_1 = \langle \Delta, T, \Sigma_1, s_{01}, abox_1, \Rightarrow_1 \rangle$ be a KB transition system, and $\mathcal{T}_2 = \langle \Delta, T_{cx}, \Sigma_2, s_{02}, abox_2, ctx, \Rightarrow_2 \rangle$ be a context-sensitive transition system, a state $s_1 \in \Sigma_1$ is *S7-bisimilar* to $s_2 \in \Sigma_2$, written $s_1 \sim_{S7} s_2$, if there exists an S7-bisimulation relation \mathcal{B} between \mathcal{T}_1 and \mathcal{T}_2 such that $\langle s_1, s_2 \rangle \in \mathcal{B}$. A transition system \mathcal{T}_1 is *S7-bisimilar* to \mathcal{T}_2 , written $\mathcal{T}_1 \sim_{S7} \mathcal{T}_2$, if there exists an S7-bisimulation relation \mathcal{B} between \mathcal{T}_1 and \mathcal{T}_2 such that $\langle s_{01}, s_{02} \rangle \in \mathcal{B}$.

We now proceed to show that two S7-bisimilar transition system can not be distinguished by $\mu\mathcal{L}_A^{EQL}$ formula modulo the translation t_{sba} .

Lemma 8.58. *Consider a KB transition system \mathcal{T}_1 and a context-sensitive transition system \mathcal{T}_2 such that $\mathcal{T}_1 \sim_{S7} \mathcal{T}_2$. For every closed $\mu\mathcal{L}_A^{EQL}$ formula Φ , we have:*

$$\mathcal{T}_1 \models \Phi \text{ if and only if } \mathcal{T}_2 \models t_{sba}(\Phi)$$

Proof. The proof is similar to the proof of Lemmas 5.42 and 6.47. The only different is that here we duplicate the modal operators eight times. All required supporting lemmas can be also easily recast into this case. \square

8.3.3 From Standard GKABS into B-AGKABS

To show that we can reduce the verification of S-GKABS into B-AGKABS, we first show that the transition system of an S-GKAB is S7-bisimilar to the transition system of its corresponding B-AGKABS that is obtained via τ_{sba} .

Lemma 8.59. *Given an S-GKAB \mathcal{G} , we have $\mathcal{R}_{\mathcal{G}}^{fs} \sim_{S7} \mathcal{R}_{\tau_{sba}(\mathcal{G})}^{f_{B}^{cx}}$*

Proof. The proof is similar to the proof of Lemma 7.46. The important difference is that, compare to B-CSGKABS, B-AGKABS basically elaborate each source of non-determinism therefore, there are seven intermediate states. \square

Last, we close this tour by showing that the verification of $\mu\mathcal{L}_A^{EQL}$ properties over S-GKAB can be recast as verification over B-AGKAB as follows.

Theorem 8.60. *Verification of closed $\mu\mathcal{L}_A^{EQL}$ properties over S-GKABS can be recast as verification over B-AGKABS.*

Proof. By Lemma 8.59, we have that $\mathcal{R}_{\mathcal{G}}^{fs} \sim_{S7} \mathcal{R}_{\tau_{sba}(\mathcal{G})}^{f_{B}^{cx}}$. Hence, by Lemma 8.58, for every $\mu\mathcal{L}_A^{EQL}$ property Φ , we have that

$$\mathcal{R}_{\mathcal{G}}^{fs} \models \Phi \text{ if and only if } \mathcal{R}_{\tau_{sba}(\mathcal{G})}^{f_{B}^{cx}} \models t_{sba}(\Phi)$$

Therefore, by using the translation τ_{sba} we can easily transform an S-GKAB into a B-AGKAB and then the claim is easily follows due to the fact above. \square

8.4 Discussion: Connection between Inconsistency-aware Context-sensitive GKABS and AGKABS

Notice that the crucial difference between AGKABS and I-CSGKABS (i.e., B-CSGKABS, C-CSGKABS, E-CSGKABS) is that I-CSGKABS wrap several non-determinism sources into a single transition while AGKABS separate them into several transitions. Thus, it is easy to see that we can easily reduce the verification of

B-CSGKABs, C-CSGKABs, and E-CSGKABs into the corresponding verification of B-AGKABs, C-AGKABs, and E-AGKABs by simply quadruplicating the modal operator of $\mu\mathcal{L}_{\text{CTX}}$ properties to be verified over I-CSGKABs (I.e., we translate $\langle-\rangle\Phi$ into $\langle-\rangle\langle-\rangle\langle-\rangle\langle-\rangle\Phi$ and similarly for $[-]$) .

SEMANTICALLY-ENHANCED DATA-AWARE PROCESSES (SEDAP_S)

As we have seen, Data-centric Dynamic Systems (DCDSs) is built based on relational database technology. In a DCDS, processes operate over the data of the system and evolve it by executing actions that may issue calls to external services. On the other hand, the (Golog) Knowledge and Action Bases is a work in a form of DCDS but based on ontologies, i.e., the data layer is represented in a rich ontology formalism, and actions perform a form of instance level update of the ontology. The use of an ontology allows for a high-level conceptual view of the data layer that is better suited for a business level treatment of the manipulated information.

Here we introduce Semantically-Enhanced Data-Aware Processes (SEDAPs), in which we merge these two approaches by enhancing a *relational layer* constituted by a DCDS-based system, with an ontology, constituting a *semantic layer*. This provides a mechanism to semantically enhance the existing data-aware processes system that is built based on relational database technology. Essentially, in SEDAPs, the ontology captures the domain of interest in which a SEDAP is executed. Additionally, it allows for seeing the data and their manipulation at a conceptual level through an ontology-based data access (OBDA) system [53, 160], reflecting the relevant concepts and relations of the domain of interest and abstracting away from how processes and data are concretely realized and stored at the concrete implementation level. It also provides us with a way of *semantically governing* the underlying DCDS-based system through the semantic layer by enabling us to specify semantic constraints at the conceptual level. Those constraints will prevent those actions that are executed at the relational layer and would lead to new system states that violates some constraints. This setting, in turn, is the basis for different important reasoning task such as verifying the evolving system through the conceptual level. Specifically, a SEDAP is constituted by three main components:

1. an *OBDA system* [53] which includes (the intensional level of) an ontology, a relational database schema, and a mapping between the ontology and the database. Essentially, it keeps all the data of interest and provides a conceptual view over it.
2. a *process component*, which characterizes the evolution of the system in the relational layer.
3. a *database instance*, which stores the initial data of the system that will be manipulated by the process component.

In the following, we use $DL-Lite_A$ for expressing ontologies and we also distinguish between objects and values. We make use a countably infinite set \mathcal{V} to denote all possible values in the system. Additionally, we consider a finite set of distinguished values $\mathcal{V}_0 \subset \mathcal{V}$. Note that the databases store values while in the ontological level, the instance of concepts are objects. Thus, similar to [53, 151], to represent the objects, we

make use a set Λ of function symbols, each with an associated arity and it also contains a special function symbol $val/1$ (that will be used to wrap values). The objects then are represented as terms of the form $f(d_1, \dots, d_n)$ where $f \in \Lambda$ and $d_1, \dots, d_n \in \mathcal{V}$. Such kind of terms are called *object terms*. We then also define the set Δ of constants as the union of \mathcal{V} and the set $\{f(d_1, \dots, d_n) \mid f \in \Lambda \text{ and } d_1, \dots, d_n \in \mathcal{V}\}$ of object terms. Last but not least, we also consider a finite set \mathcal{F} of *function symbols* that represents *service calls*, which abstractly account for the injection of fresh values (constants) from Δ into the system.

The results in this chapter are published in [57, 58, 164, 26, 25]

9.1 Formalizing SEDAPs

Before we formally defined SEDAPs in Section 9.1.2, we first briefly review the Ontology Based Data Access (OBDA) in Section 9.1.1 to give some necessary preliminaries.

9.1.1 Ontology Based Data Access (OBDA) at a glance

In an Ontology Based Data Access (OBDA) system, a relational database is connected to an ontology that represents the domain of interest by a mapping, which relates database values with values and (abstract) objects in the ontology (c.f. [53, 151]). The mapping in OBDA is formally defined as follows:

OBDA Mapping **Definition 9.1** (OBDA Mapping). Given a TBox T , and a database schema \mathcal{R} , an *OBDA mapping* \mathcal{M} over T and \mathcal{R} is a set of mapping assertions, each of the form

$$\Phi(\vec{x}) \rightsquigarrow \Psi(\vec{t}),$$

where:

1. \vec{x} is a non-empty set of variables,
2. \vec{t} is a set of terms of the form $f(\vec{z})$, with $f \in \Lambda$ and $\vec{z} \subseteq \vec{x}$,
3. $\Phi(\vec{x})$ is an SQL query over \mathcal{R} , with \vec{x} as output variables (Note that we only consider the core SQL fragment that corresponds to DI-FOL), and
4. $\Psi(\vec{t})$ is a CQ over T without non-distinguished variables, whose atoms are the terms \vec{t} .

Without loss of generality, we use a special function symbol $val/1$ to map values from the relational layer to the range of attributes in the semantic layer. ■

Formally, an OBDA systems is then defined as follows.

OBDA System **Definition 9.2** (OBDA System). Formally, an *OBDA system* is a structure $\mathcal{O} = \langle T, \mathcal{R}, \mathcal{M} \rangle$, where:

1. T is a *DL-Lite_A* TBox;
 2. $\mathcal{R} = \{R_1, \dots, R_n\}$ is a database schema, constituted by a finite set of relation schemas;
 3. \mathcal{M} is an OBDA mapping over T and \mathcal{R} .
-

Example 9.3. For the running example of this chapter, recall the simple order processing scenario that is used in Example 2.54. We now specify an OBDA system $\mathcal{O} = \langle T, \mathcal{R}, \mathcal{M} \rangle$ where

- T is the same as the TBox specified in Example 2.17.
- \mathcal{R} is the same as the database schema specified as in Example 2.6.
- \mathcal{M} contains the following mapping assertions:

```

 $m_1$  : SELECT id, name FROM ORDER
      WHERE processing_status = "approved"
       $\rightsquigarrow$  ApprovedOrder ( $ord(id, name)$ )

 $m_2$  : SELECT id, name FROM ORDER
      WHERE processing_status = "received"
       $\rightsquigarrow$  ReceivedOrder ( $ord(id, name)$ )

 $m_3$  : SELECT id, name FROM ORDER
      WHERE processing_status = "assembled"
       $\rightsquigarrow$  AssembledOrder ( $ord(id, name)$ )

 $m_4$  : SELECT id, name FROM ORDER o, DELIVERED_ORDER d,
      WHERE o.id = d.id
       $\rightsquigarrow$  DeliveredOrder ( $ord(id, name)$ )

 $m_5$  : SELECT assembler FROM ORDER
      WHERE assembler IS NOT NULL
       $\rightsquigarrow$  Assembler ( $emp(assembler)$ )

 $m_6$  : SELECT designer FROM ORDER
      WHERE designer IS NOT NULL
       $\rightsquigarrow$  Designer ( $emp(designer)$ )

 $m_7$  : SELECT quality_controller FROM ORDER
      WHERE quality_controller IS NOT NULL
       $\rightsquigarrow$  QualityController ( $emp(quality\_controller)$ )

 $m_8$  : SELECT id, name, assembling_loc FROM ORDER
      WHERE assembling_loc IS NOT NULL
       $\rightsquigarrow$  hasAssemblingLoc ( $ord(id, name), loc(assembling\_loc)$ )

 $m_9$  : SELECT id, name, design FROM ORDER
      WHERE design IS NOT NULL
       $\rightsquigarrow$  hasDesign ( $ord(id, name), des(design)$ )

 $m_{10}$  : SELECT id, name, assembler FROM ORDER
      WHERE assembler IS NOT NULL
       $\rightsquigarrow$  assembledBy ( $ord(id, name), emp(assembler)$ )

```

```

 $m_{11}$  : SELECT id, name, designer FROM ORDER
        WHERE designer IS NOT NULL
         $\rightsquigarrow$  designedBy ( $ord(id, name)$ ,  $emp(designer)$ )
 $m_{12}$  : SELECT id, name, quality_controller FROM ORDER
        WHERE quality_controller IS NOT NULL
         $\rightsquigarrow$  checkedBy ( $ord(id, name)$ ,  $emp(quality\_controller)$ )

```

The intuition of the mappings above is as follows: the mapping m_1 (resp. m_2 and m_3) maps every order in ORDER with processing_status “approved” (resp. “received” and “assembled”) to an ApprovedOrder (resp. ReceivedOrder and AssembledOrder). Such an order is constructed by “objectifying” the id and name using function $ord/2$. The mapping m_4 generates delivered order by selecting only those orders in the ORDER table whose id is also contained in the DELIVERED_ORDER table. The mapping m_5 (resp. m_6 and m_7) populates the concept Assembler (resp. Designer and QualityController) with the assembler (resp. designer and quality_controller) data in ORDER. The role hasAssemblingLoc (resp. hasDesign) is populated by the mapping m_8 (resp. m_9) using the data about orders in ORDER and their corresponding assembling location (resp. design). Finally, the mapping m_{10} (resp. m_{11} and m_{12}) populate the role assembledBy (resp. designedBy and checkedBy) with the orders in ORDER and their corresponding assembler (resp. designer and quality controller).

Given a database instance \mathbf{I} over a database schema \mathcal{R} with $\text{ADOM}(\mathbf{I}) \subseteq \mathcal{V}$, and given a mapping \mathcal{M} , the *virtual ABox* generated from \mathbf{I} by a mapping assertion $m = \Phi(\vec{x}) \rightsquigarrow \Psi(\vec{t})$ in \mathcal{M} is $m(\mathbf{I}) = \bigcup_{\sigma \in \text{ANS}(\Phi, \mathbf{I})} \Psi\sigma$. Then, the *virtual ABox* generated from \mathbf{I} by the mapping \mathcal{M} is $\mathcal{M}(\mathbf{I}) = \bigcup_{m \in \mathcal{M}} m(\mathbf{I})$. Notice that $\text{ADOM}(\mathcal{M}(\mathbf{I})) \subseteq \Delta$. Given an OBDA system $\mathcal{O} = \langle T, \mathcal{R}, \mathcal{M} \rangle$ and a database instance \mathbf{I} over a database schema \mathcal{R} , a *model for \mathcal{O} wrt \mathbf{I}* is an interpretation \mathcal{I} such that $\mathcal{I} \models (T, \mathcal{M}(\mathbf{I}))$. We say that \mathcal{O} is satisfiable w.r.t. \mathbf{I} if it admits at least one model w.r.t. \mathbf{I} .

Example 9.4. Continuing our running example (see Example 9.3), consider a database instance

$$\mathbf{I} = \{\text{ORDER}(123, \text{chair}, \text{approved}, 456, \text{alice}, \text{bob}, \text{john}, \text{bolzano}, \text{ecodesign})\}.$$

The corresponding virtual ABox obtained from the application of the mapping \mathcal{M} is

$$\begin{aligned} \mathcal{M}(\mathbf{I}) = \{ & \text{ApprovedOrder}(ord(123, \text{chair})), \text{Assembler}(emp(\text{bob})), \\ & \text{Designer}(emp(\text{alice})), \text{Quality_Controller}(emp(\text{john})), \\ & \text{hasAssemblingLoc}(ord(123, \text{chair}), loc(\text{bolzano})), \\ & \text{hasDesign}(ord(123, \text{chair}), des(\text{ecodesign})), \\ & \text{assembledBy}(ord(123, \text{chair}), emp(\text{bob})), \\ & \text{designedBy}(ord(123, \text{chair}), emp(\text{alice})), \\ & \text{checkedBy}(ord(123, \text{chair}), emp(\text{john})) \} \end{aligned}$$

A UCQ q over an OBDA system $\mathcal{O} = \langle T, \mathcal{R}, \mathcal{M} \rangle$ is simply a UCQ over T . To compute the certain answers of q over \mathcal{O} wrt a database instance \mathbf{I} for \mathcal{R} , we follow a three-step approach:

1. q is *rewritten* to compile away T , obtaining $q_r = \text{rew}(q, T)$;
2. the mapping \mathcal{M} is used to *unfold* q_r into a query over \mathcal{R} , denoted by $\text{UNFOLD}(q_r, \mathcal{M})$, which turns out to be an SQL query [151];
3. such a query is then evaluated over \mathbf{I} , obtaining the certain answers.

For an ECQ, we can proceed in a similar way, applying the rewriting and unfolding steps to the embedded UCQs. It follows that computing certain answers to UCQs/ECQs in an OBDA system is FO rewritable. Furthermore, applying the unfolding step to Q_{unsatFOL}^T , we obtain also that satisfiability in \mathcal{O} can be reduced into evaluating a query over \mathbf{I} .

9.1.2 Formalization of SEDAPs

Roughly speaking, a SEDAP is constituted by: (i) A *Relational Layer*, which captures the database evolution (manipulation) by actions. (ii) A *Semantic Layer*, which exploits the ontology for providing a conceptual view of the system evolution. (iii) A set of *mapping assertions* describing how to virtually project data concretely maintained at the Relational Layer into concepts and relations modeled in the Semantic Layer, thus providing a link between the data in the relational layer and the ontology.

To avoid unnecessary complication, here we only use a set of condition-action rules (c.f. Definition 2.51) to formalize the progression mechanism that evolves the data in the relational layer. However, one can easily lift it into a more complex/sophisticated formalism such as Golog program similar to Chapter 4, but this is not our focus on this chapter. Moreover, here we use the DCDS actions formalisms, and we will also see later that we use the DCDS actions execution semantics that rebuild the whole database instance at each action execution. However, it is easy to see that actually we can change the setting such that it uses the KAB actions formalisms (with a slight modification on the queries) and KAB actions execution semantics at no cost. This is possible due to the fact that we can simulate the KAB actions execution semantics in DCDS actions execution semantics as we have seen when we compile KABs into DCDSs in Section 3.3.1. One reason why we use DCDS actions and their execution semantics here is because we want to simplify the proof and avoid unnecessary complication while focusing on presenting the setting.

Formally, a SEDAP is then defined as follows:

Definition 9.5 (SEDAP). A *SEDAP* \mathcal{S} is a tuple $\langle T, \mathcal{R}, \mathcal{M}, \mathbf{I}_0, \mathcal{A}, \varrho \rangle$, where:

- T is a *DL-Lite_A* TBox,
- \mathcal{R} is a database schema,
- \mathcal{M} is an OBDA mapping over T and \mathcal{R} (see Definition 9.1),
- \mathbf{I}_0 is a database instance over \mathcal{R} ,
- \mathcal{A} is a set of DCDS actions over \mathcal{R} and \mathbf{I}_0 (see Definition 2.50),
- ϱ is a set of DCDS condition-action rules over \mathcal{R} , \mathbf{I}_0 , and \mathcal{A} (see Definition 2.51).

Semantically-Enhanced Data-Aware Processes (SEDAP)

■

Together T , \mathcal{R} , and \mathcal{M} constitute the OBDA system, while \mathcal{A} and ϱ form the *process component*.

Example 9.6. Consider our running example, recall the OBDA system $\mathcal{O} = \langle T, \mathcal{R}, \mathcal{M} \rangle$ in Example 9.3. To model our simple order processing scenario, we specify a SEDAP $\mathcal{S} = \langle T, \mathcal{R}, \mathcal{M}, \mathbf{I}_0, \mathcal{A}, \varrho \rangle$ where

- T, \mathcal{R} , and \mathcal{M} are the same as in Example 9.3 (i.e., T is specified in Example 2.17, and \mathcal{R} is specified in Example 2.6).
- \mathcal{A} (resp. ϱ) is the same as the set of actions (resp. condition-action rules) specified in Example 2.54.
- The initial database \mathbf{I}_0 is specified as follows:

$$\begin{aligned} \mathbf{I}_0 = \{ & \text{ORDER}(123, \text{chair}, \text{received}, 456, \text{NULL}, \text{NULL}, \\ & \text{NULL}, \text{NULL}, \text{ecodesign}), \\ & \text{ORDER}(321, \text{table}, \text{approved}, 654, \text{NULL}, \text{NULL}, \\ & \text{NULL}, \text{NULL}, \text{NULL}) \} \end{aligned}$$

9.2 SEDAPs Execution Semantics

The semantics of SEDAPs is provided in terms of possibly infinite transition systems. More specifically, two transition systems are constructed to describe the execution semantics of SEDAPs:

1. A *Relational Layer Transition System* (RTS), representing all allowed computations that, starting from the initial database \mathbf{I}_0 , the process component can do over the data in the relational layer, according to the constraints imposed at the semantic layer (semantic governance).
2. A *Semantic Layer Transition System* (STS), representing the same computations at the semantic layer (abstracting the evolution in the relational layer).

Both of them are formally defined as follows:

*Relational Layer
Transition System
(RTS)*

Definition 9.7 (Relational Layer Transition System (RTS)). Given a SEDAP $\mathcal{S} = \langle T, \mathcal{R}, \mathcal{M}, \mathbf{I}_0, \mathcal{A}, \varrho \rangle$, we define the *Relational Layer Transition System* (RTS) of \mathcal{S} , written \mathcal{R}_S^R , as a tuple $\langle \Delta, \mathcal{R}, \Sigma, s_0, db, \Rightarrow \rangle$, where:

1. Σ is a set of states where each state $s \in \Sigma$ is defined as a tuple $\langle \mathbf{I}, m \rangle$, where \mathbf{I} is a database instance and m is a service call map,
2. s_0 is an initial state,
3. db is a function that, given a state in \mathcal{R}_S^R , returns a corresponding database instance (conforming to \mathcal{R}),
4. $\Rightarrow \subseteq \Sigma \times \Sigma$ is the transition relation.

The components Σ, \Rightarrow and db of \mathcal{R}_S^R are defined by simultaneous induction as the smallest sets satisfying the following conditions:

- $s_0 = \langle \mathbf{I}_0, \emptyset \rangle \in \Sigma$, $db(s_0) = \mathbf{I}_0$;
- if $s = \langle \mathbf{I}, m \rangle \in \Sigma$, then for all actions $\alpha \in \mathcal{A}$, for all legal parameter assignments σ for α in \mathbf{I} and for all $\langle \mathbf{I}', m' \rangle$ such that
 1. $\langle \mathbf{I}, m \rangle \xrightarrow{\alpha\sigma, \mathcal{S}} \langle \mathbf{I}', m' \rangle$, and
 2. the OBDA system $\mathcal{O} = \langle T, \mathcal{R}, \mathcal{M} \rangle$ is *satisfiable* w.r.t. \mathbf{I}' .
 we have $s' = \langle \mathbf{I}', m' \rangle \in \Sigma$, and $s \Rightarrow s'$.

(Note that the relation $\xrightarrow{\alpha\sigma, \mathcal{S}}$ is defined in Definition 2.59). ■

Observe that the satisfiability check that is done in the last step of the RTS construction realizing the notion of *semantic governance*. I.e., the system evolution in the relational layer taking into account the constraints specified in the semantic layer.

The semantic layer transition system (STS) \mathcal{Y}_S^S of a SEDAP \mathcal{S} is basically a “virtualization” of the RTS \mathcal{Y}_S^R of \mathcal{S} in the semantic layer. It is basically obtained from the RTS \mathcal{Y}_S^R . Essentially, STS maintains the structure of \mathcal{Y}_S^R unaltered, reflecting that the process component is executed over the relational layer, but it associates each state s to a virtual ABox obtained from the application of the mapping \mathcal{M} to the database instance associated by \mathcal{Y}_S^R to the same state s . Formally, the STS is then defined as follows:

Definition 9.8 (Semantic Layer Transition System (STS)). Given a SEDAP $\mathcal{S} = \langle T, \mathcal{R}, \mathcal{M}, \mathbf{I}_0, \mathcal{A}, \varrho \rangle$, let $\mathcal{Y}_S^R = \langle \Delta, \mathcal{R}, \Sigma, s_0, db, \Rightarrow \rangle$ be its RTS. We define the *Semantic Layer Transition System (STS)* of \mathcal{S} , written \mathcal{Y}_S^S , as a tuple $\langle \Delta, T, \Sigma, s_0, abox, \Rightarrow \rangle$ such that for each $s \in \Sigma$, $abox(s) = \mathcal{M}(db(s))$. (Note that Σ , s_0 , and \Rightarrow are the same as in \mathcal{Y}_S^R). ■

*Semantic Layer
Transition System
(STS)*

The intuition of the SEDAP setting (the semantic layer transition system and the relational layer transition system) is depicted in Figure 5.

Example 9.9. Continuing our running example in Example 9.6 and reconsider the SEDAP \mathcal{S} specified in that example. The construction of RTS \mathcal{Y}_S^R is started from the initial state $s_0 = \langle \mathbf{I}_0, \emptyset \rangle$, where:

$$\mathbf{I}_0 = \{ \text{ORDER}(123, \text{chair}, \text{received}, 456, \text{NULL}, \text{NULL}, \text{NULL}, \text{NULL}, \text{ecodesign}), \\ \text{ORDER}(321, \text{table}, \text{approved}, 654, \text{NULL}, \text{NULL}, \text{NULL}, \text{NULL}, \text{NULL}) \}$$

An example of a successor of state s_0 is a state $s_1 = \langle \mathbf{I}_1, m_1 \rangle$, where

$$\mathbf{I}_1 = \{ \text{ORDER}(123, \text{chair}, \text{approved}, 456, \text{NULL}, \text{NULL}, \text{NULL}, \text{NULL}, \text{ecodesign}), \\ \text{ORDER}(321, \text{table}, \text{approved}, 654, \text{NULL}, \text{NULL}, \text{NULL}, \text{NULL}, \text{NULL}) \},$$

and s_1 is obtained similarly as in Example 2.61. Note that we use the same set of actions and condition-action rules as in Example 2.61. The construction of \mathcal{Y}_S^R is continued further by applying all possible actions and so on.

On the other hand, the STS \mathcal{Y}_S^S is obtained by virtualizing \mathcal{Y}_S^R into the semantic layer by utilizing the mapping \mathcal{M} . The corresponding ABox of each state is obtained by applying mapping \mathcal{M} to the database instance of the corresponding states. For instance, we have that

$$abox(s_0) = \mathcal{M}(db(s_0)) = \{ \text{ReceivedOrder}(\text{ord}(123, \text{chair})), \\ \text{hasDesign}(\text{ord}(123, \text{chair}), \text{des}(\text{ecodesign})), \\ \text{ApprovedOrder}(\text{ord}(321, \text{table})) \}$$

and also a successor of s_0 is s_1 where

$$\begin{aligned} \text{abox}(s_1) = \mathcal{M}(\text{db}(s_1)) = \{ & \text{ApprovedOrder}(\text{ord}(123, \text{chair})), \\ & \text{hasDesign}(\text{ord}(123, \text{chair}), \text{des}(\text{ecodesign})), \\ & \text{ApprovedOrder}(\text{ord}(321, \text{table}))\} \end{aligned}$$

9.3 Correspondence Between SEDAPs and DCDSs

The interesting task in SEDAPs is to verify the compliance of SEDAPs evolution against the conceptual temporal properties specified over the semantic layer. To tackle this issue, later we will see that the verification of SEDAPs can be reduced to the verification of DSDSs and hence we can take the advantages from the well-established results in DCDSs [24]. To this aim, in this section we establish an interesting correspondence between SEDAPs and DCDSs. In particular, here we present the mechanism of compiling SEDAPs into DCDSs.

We now define a translation ς that takes a SEDAP \mathcal{S} as input and produce a DCDS $\varsigma(\mathcal{S})$ such that the transition system $\mathcal{T}_{\varsigma(\mathcal{S})}$ of $\varsigma(\mathcal{S})$ is equivalent to the relational transition system $\mathcal{T}_{\mathcal{S}}^R$ of \mathcal{S} .

*Translation From
SEDAP to DCDS*

Definition 9.10 (Translation From SEDAP to DCDS). We define a translation ς that, given a SEDAP $\mathcal{S} = \langle T, \mathcal{R}, \mathcal{M}, \mathbf{I}_0, \mathcal{A}, \varrho \rangle$, produces a DCDS $\varsigma(\mathcal{S}) = \langle D, P \rangle$ such that

- $D = \langle \mathcal{R}, \mathbf{I}_0, \mathcal{E} \rangle$ is a DCDS data component with $\mathcal{E} = \{\text{UNFOLD}(Q_{\text{unsatFOL}}^T, \mathcal{M}) \rightarrow \text{false}\}$, where Q_{unsatFOL}^T is an FOL query defined in Definition 2.43. Intuitively, we encode the constraints in the TBox T into the equality constraints \mathcal{E} in DCDS.
- $P = \langle \mathcal{A}, \varrho \rangle$ is a DCDS process component over D .

■

Essentially, to obtain a DCDS from a SEDAP, we compile the negative inclusion and functionality assertions in the TBox into equality constraints. Thus, roughly speaking we delegate the consistency check into the relational layer. Having the translation ς in hand, we can easily show the following theorem.

Theorem 9.11. *Given a SEDAP \mathcal{S} with RTS $\mathcal{T}_{\mathcal{S}}^R$, let $\varsigma(\mathcal{S})$ be its corresponding DCDS obtained via ς and $\mathcal{T}_{\varsigma(\mathcal{S})}$ be the corresponding TS of $\varsigma(\mathcal{S})$. We have that $\mathcal{T}_{\mathcal{S}}^R$ is equivalent to $\mathcal{T}_{\varsigma(\mathcal{S})}$*

Proof. Let $\mathcal{S} = \langle T, \mathcal{R}, \mathcal{M}, \mathbf{I}_0, \mathcal{A}, \varrho \rangle$ and $\varsigma(\mathcal{S}) = \langle D, P \rangle$ where $D = \langle \mathcal{R}, \mathbf{I}_0, \mathcal{E} \rangle$, $\mathcal{E} = \{\text{UNFOLD}(Q_{\text{unsatFOL}}^T, \mathcal{M}) \rightarrow \text{false}\}$, and $P = \langle \mathcal{A}, \varrho \rangle$. The proof can be easily obtained by considering the following:

- Both \mathcal{S} and $\varsigma(\mathcal{S})$ start from the same initial database instance \mathbf{I}_0 .
- The satisfiability check of an OBDA system $\mathcal{O} = \langle T, \mathcal{R}, \mathcal{M} \rangle$ w.r.t. a database instance \mathbf{I} can be delegated to checking whether \mathbf{I} satisfy \mathcal{E} due to the correctness of rewriting and unfolding procedure in answering queries over an OBDA system [151] and the fact that the satisfiability check in $DL\text{-}Lite_{\mathcal{A}}$ can be delegated into query answering [50]. Thus, it follows that for each state s_s in $\mathcal{T}_{\mathcal{S}}^R$ and state s_d

in $\Upsilon_{\varsigma(\mathcal{S})}$ such that $db(s_s) = db(s_d)$, s_s is a consistent state if and only if s_d is a consistent state.

- both Υ_{ς}^R and $\Upsilon_{\varsigma(\mathcal{S})}$ has the same structure since their process component are the same and each corresponding states contain the same database instance. \square

9.4 Verifying SEDAPs

Given a SEDAP \mathcal{S} , we are interested in studying the verification of semantic temporal properties specified over the Semantic Layer. Technically, this means that properties are verified against the SEDAP's STS $\Upsilon_{\mathcal{S}}^S$. Moreover, the temporal properties to be verified combines temporal operators with queries posed over the ontologies obtained by combining the TBox T with the ABoxes associated to the states of $\Upsilon_{\mathcal{S}}^S$. As verification formalisms, here we consider $\mu\mathcal{L}_A^{\text{EQL}}$. The verification problem of $\mu\mathcal{L}_A^{\text{EQL}}$ over SEDAPs is then formally defined as follows:

Definition 9.12 (Verification of $\mu\mathcal{L}_A^{\text{EQL}}$ over SEDAPs). Given a SEDAP \mathcal{S} and a $\mu\mathcal{L}_A^{\text{EQL}}$ formula Φ , the *verification of Φ over \mathcal{S}* is the problem of checking whether $\Upsilon_{\mathcal{S}}^S \models \Phi$, where $\Upsilon_{\mathcal{S}}^S$ is the semantic layer transition system of \mathcal{S} . \blacksquare

*Translation From
SEDAP to DCDS*

The problem is that the temporal properties are specified over the semantic layer but the system actually evolves at the relational layer. So, to reconcile these pieces, we bring the verification down to the relational layer. We show that verification of $\mu\mathcal{L}_A^{\text{EQL}}$ properties over the STS $\Upsilon_{\mathcal{S}}^S$ can be reduced to verification of $\mu\mathcal{L}_A$ properties over the corresponding RTS $\Upsilon_{\mathcal{S}}^R$.

The reduction is realized by providing a translation mechanism from Φ into a corresponding $\mu\mathcal{L}_A$ property Φ' specified over \mathcal{R} , and then showing that $\Upsilon_{\mathcal{S}}^S \models \Phi$ if and only if $\Upsilon_{\mathcal{S}}^R \models \Phi'$. This translation is based on the notion of rewriting and unfolding as the procedure to compute the certain answers over an OBDA system. Given a SEDAP $\mathcal{S} = \langle T, \mathcal{R}, \mathcal{M}, \mathbf{I}_0, \mathcal{A}, \varrho \rangle$ and a $\mu\mathcal{L}_A^{\text{EQL}}$ formula Φ , the general strategy about the translation for Φ is as follows:

1. We keep the temporal part unaltered.
2. We rewrite each query in Φ w.r.t. the TBox T in order to compile away the TBox and incorporate the knowledge encoded in T . Formally this step is defined in Definition 3.20. Essentially this step transforms each query in Φ into a DI-FOL query.
3. We unfold the rewritten temporal formula $rew(\Phi, T)$ based on the given mapping \mathcal{M} in order to transform each rewritten query in Φ into a query over \mathcal{R} .

An important observation while unfolding the query is that in the semantic layer the elements of the active domain are objects while in the relational layer the elements of the active domain are values. Hence, we also need to transform the quantification over the objects into the quantification of the corresponding values that form the objects. The unfolding mechanism for the rewritten temporal formula is formally defined as follows:

Unfolding Mechanism
for $\mu\mathcal{L}_A^{\text{EQL}}$

Definition 9.13 (Unfolding Mechanism for $\mu\mathcal{L}_A^{\text{EQL}}$). Given a TBox T , a database schema \mathcal{R} , a $\mu\mathcal{L}_A^{\text{EQL}}$ formula Φ over T , a mapping \mathcal{M} over T and \mathcal{R} . Let $\Phi' = \text{rew}(\Phi, T)$ be the rewritten formula of Φ w.r.t. T , we define the unfolding of Φ' w.r.t. \mathcal{M} , written $\text{UNFOLD}(\Phi', \mathcal{M})$, recursively as follows:

$$\text{UNFOLD}(\Phi', \mathcal{M}) = \begin{cases} \text{UNFOLD}(Q, \mathcal{M}) & \text{if } \Phi' = Q \\ \text{UNFOLD}(\Psi_1, \mathcal{M}) \vee \text{UNFOLD}(\Psi_2, \mathcal{M}) & \text{if } \Phi' = \Psi_1 \vee \Psi_2 \\ \bigvee_{(f/n) \in \text{FS}(\mathcal{M})} \exists x_1, \dots, x_n. \text{UNFOLD}(\Psi[x/f(x_1, \dots, x_n)], \mathcal{M}) & \text{if } \Phi' = \exists x. \Psi \\ \langle \neg \rangle \text{UNFOLD}(\Psi, \mathcal{M}) & \text{if } \Phi' = \langle \neg \rangle \Psi \\ \mu Z. \text{UNFOLD}(\Psi, \mathcal{M}) & \text{if } \Phi' = \mu Z. \Psi \end{cases}$$

where:

1. $\text{UNFOLD}(Q, \mathcal{M})$ is as in the usual unfolding in OBDA (c.f. [151, 53]).
2. $\text{FS}(\mathcal{M})$ is the set of function symbols that are used to form the object terms and occur in \mathcal{M} (including the special function symbol $\text{val}/1$).

■

For unfolding the query, we unfold the query of the form $\exists x.Q'$ as follows:

$$\text{UNFOLD}(\exists x.Q', \mathcal{M}) =$$

$$\bigvee_{(f/n) \in \text{FS}(\mathcal{M})} \exists x_1, \dots, x_n. \text{UNFOLD}(Q'[x/f(x_1, \dots, x_n)], \mathcal{M}).$$

I.e., we unfold $\exists x.Q'$ into a disjunction of formulas, where each formula is obtained from Q by replacing x with one of the possible terms constructed from function symbols in \mathcal{M} , and then existentially quantify each variable that form the corresponding term. The reason of this unfolding is to rephrase the quantification over object terms into the corresponding quantification over values in the relational layer that could lead to produce such object terms and values through the application of \mathcal{M} . This is done by unfolding $\exists x.Q$ into a disjunction of formulas, where each of the formula is obtained from Q by replacing x with one of the possible variable terms constructed from function symbols in \mathcal{M} , and quantifying over the existence of values that could form a corresponding object term.

For unfolding the UCQ, the atoms in the UCQ are unified with the heads of the mapping assertions in \mathcal{M} . For each successful unification, each atom is replaced with the body of the corresponding mapping. The unfolding of the UCQ is then obtained as the union of all queries obtained in this way. Other cases of query are simply managed by pushing the unfolding down to the sub-formulas.

Theorem 9.14. Let $\mathcal{S} = \langle T, \mathcal{R}, \mathcal{M}, \mathbf{I}_0, \mathcal{A}, \varrho \rangle$ be a SEDAP, $\gamma_S^R = \langle \Delta, \mathcal{R}, \Sigma, s_0, db, \Rightarrow \rangle$ and $\gamma_S^S = \langle \Delta, T, \Sigma, s_0, abox, \Rightarrow \rangle$ consecutively be the RTS and STS of \mathcal{S} . Consider a $\mu\mathcal{L}_A^{\text{EQL}}$ formula Φ over T . Then:

$$\gamma_S^S \models \Phi \quad \text{if and only if} \quad \gamma_S^R \models \text{UNFOLD}(\text{rew}(\Phi, T), \mathcal{M})$$

Proof. The proof can be simply obtained by observing the following:

1. Both \mathcal{R}_S^S and \mathcal{R}_S^R have the same structure. In fact, essentially they can be seen as a single transition system such that each state s in the transition system has its own associated database instance and ABox and $abox(s) = \mathcal{M}(db(s))$.
2. The unfolding and the rewriting process do not alter the temporal part of the formula.
3. The correctness of local queries is obtained by the correctness of the unfolding and rewriting in OBDA (see [151]).
4. The obtained formula $\text{UNFOLD}(\text{rew}(\Phi, T), \mathcal{M})$ is a $\mu\mathcal{L}_A$ property that can be verified over RTS (note that RTS is the same as database transition systems that define the semantics of $\mu\mathcal{L}_A$).

□

Due to the injection of new, fresh data into the system due to call to external services, \mathcal{R}_S^R (as well as \mathcal{R}_S^S) is in general infinite-state. This causes verification to be undecidable in general, even for the very simple case of a SEDAP in which the TBox contains no assertions and directly reflects the database schema via simple one-to-one mappings. This boils down to the undecidability result of DCDS [24].

An extensive study concerning some decidability boundaries for the verification of Data-Centric Dynamic Systems (DCDSs) with non-deterministic external services has been provided in [24]. One of the interesting conditions for decidability that have been studied is *run-boundedness*. Thus, to gain decidability, we adopt such restriction and we show that the verification of run-bounded SEDAPs can be reduced to the verification of run-bounded DCDSs.

Definition 9.15 (Run-bounded SEDAP). Given a SEDAP \mathcal{S} with RTS $\mathcal{R}_S^R = \langle \Delta, T, \Sigma, s_0, db, \Rightarrow \rangle$, we say \mathcal{S} is *run-bounded* if there exists an integer bound b such that for every run $\pi = s_0 s_1 \dots$ of \mathcal{R}_S^R , we have that $|\bigcup_{s \text{ state of } \pi} \text{ADOM}(db(s))| < b$. ■

Run-bounded SEDAP

Utilizing Theorem 9.14 and the well-established result for DCDS, in the following we show the decidability of the $\mu\mathcal{L}_A^{\text{EQL}}$ verification over run-bounded SEDAPs.

Theorem 9.16. *Verification of $\mu\mathcal{L}_A^{\text{EQL}}$ properties over run-bounded SEDAPs is decidable, and can be reduced to conventional finite-state model checking.*

Proof. Let \mathcal{S} be a SEDAP with RTS \mathcal{R}_S^R and STS \mathcal{R}_S^S , $\varsigma(\mathcal{S})$ be its corresponding DCDS obtained through translation ς and has transition system $\mathcal{Y}_{\varsigma(\mathcal{S})}$. By Theorem 9.11, we have that \mathcal{R}_S^R and $\mathcal{Y}_{\varsigma(\mathcal{S})}$ are equivalent. Thus, \mathcal{S} is run-bounded if and only if $\varsigma(\mathcal{S})$ is run-bounded. Furthermore, by using Theorem 9.14, since $\mathcal{R}_S^S \models \Phi$ if and only if $\mathcal{R}_S^R \models \text{UNFOLD}(\text{rew}(\Phi, T), \mathcal{M})$, it follows that $\mathcal{R}_S^S \models \Phi$ if and only if $\mathcal{Y}_{\varsigma(\mathcal{S})} \models \text{UNFOLD}(\text{rew}(\Phi, T), \mathcal{M})$ (consider also that $\text{UNFOLD}(\text{rew}(\Phi, T), \mathcal{M})$ is a $\mu\mathcal{L}_A$ formula). The proof is then completed since according to Theorem 2.65, the verification of $\mu\mathcal{L}_A$ over run-bounded DCDSs is decidable and can be reduced to conventional finite-state model checking. □

9.5 From Theory to Practice: SEDAPs Instantiation

So far we have introduced SEDAPs as a formal framework for representing data-aware processes system equipped with a Semantic Layer. In particular, we have focused our

attention to the usage of lightweight Description Logics, belonging to the DL-Lite family, to conceptually capture the relevant domain entities and relationships at the Semantic Layer. At the same time, we have shown that, thanks to the FO rewritability of DL-Lite, verification of temporal properties over the evolution of a data-aware processes system understood through the lens of the Semantic Layer can be faithfully reduced to verification of properties directly carried out at the Relational Layer.

In this section, we show how the formal framework SEDAPs can be concretely instantiated. This work is part of the deliverables of EU FP7 Project namely ACSI (“Artifact-Centric Service Interoperation”, see <http://www.acsi-project.eu/>). Specifically, the results presented here is part of the ACSI deliverable in [60]. To concretize the idea of SEDAPs, here we consider a specific setting where the transition relation at the Relational Layer is obtained from *artifacts* (*artifact-centric systems*) specified using the *Guard-Stage-Milestone* (GSM) [126, 87] approach. In particular, we exhibit how existing techniques and tools can be suitably combined into a tool, called *OBGSM*, which enables the verification of GSM-based data-aware processes system equipped with a Semantic Layer. To this aim, we leverage on the following tools:

1. -ONTOP-¹, a JAVA-based framework for OBDA, and in particular the Quest reasoner, which is the component dedicated to handle query rewriting and unfolding;
2. the *GSMC model checker*, developed within ACSI project to verify GSM-based artifact-centric systems against temporal properties [114, 113, 31].

The main purpose of OBGSM is: given a temporal property specified over the Semantic Layer of the system, together with mapping assertions whose language is suitably shaped to work with GSMC, automatically rewrite and unfold the property by producing a corresponding translation that can be directly processed by the GSMC model checker. This cannot be done by solely relying on the functionalities provided by -ONTOP-, for two reasons:

1. OBGSM deals with temporal properties specified in a fragment of $\mu\mathcal{L}_A^{\text{EQL}}$, and not just (local) ECQs;
2. the mapping assertions are shaped so as to reflect the specific query language supported by GSMC, guaranteeing that the rewriting and unfolding process produces a temporal property expressed in the input language of GSMC.

In particular, we note that GSMC is not able to process the entire $\mu\mathcal{L}_A^{\text{EQL}}$ logic, but only its CTL fragment. Here we denote such fragment by CTL_A . This requires also to restrict the $\mu\mathcal{L}_A^{\text{EQL}}$ verification formalism accordingly, in particular focusing on its CTL fragment, denoted by $\text{CTL}_A^{\text{EQL}}$.

An important observation related to semantic governance in this setting is that since the construction of the RTS for GSM is handled internally by GSMC, it is not possible (at least for the time being) to prune it so as to remove inconsistent states. Therefore, in the following we assume that all the states in the RTS are consistent with the constraints of the Semantic Layer. This can be trivially achieved by, e.g., avoiding to use negative inclusion in the TBox.

Before we proceed with the system specification of OBGSM in Section 9.5.2, we first briefly review the Guard-Stage-Milestone (GSM) in Section 9.5.1. In the following we

¹ <http://ontop.inf.unibz.it>

might use the terms artifact layer and relational layer interchangeably in order to refer to the notion of relational layer as introduced in SEDAPs.

9.5.1 Guard-Stage-Milestone (GSM) at a Glance

Guard-Stage-Milestone (GSM) [126] has been proposed as a framework for modeling/specifying artifact-centric systems [147, 125] which combine both static and dynamic aspects of the systems. In artifact-centric systems, an *artifact* is characterized by an *information model*, which maintains the artifact data, and by a *lifecycle* that specifies the allowed ways to progress the information model (i.e., characterize the evolution of the system). Among the different proposals for artifact-centric process modeling, the GSM approach has been proposed to model artifacts and their lifecycle in a declarative, flexible way. GSM is equipped with a formal execution semantics [87], which unambiguously characterizes the artifact progression in response to external events. Notably, several key constructs of the emerging OMG standard on Case Management and Model Notation² have been borrowed from GSM.

Here we only provide a general overview of the Guard-Stage-Milestone (GSM) methodology and we refer to [126, 87] for more detailed and formal definitions. Technically, a GSM model consists of a set of artifact types where each artifact type has its own information model as well as lifecycle and when an artifact type is instantiated, its instance has a corresponding identifier. During the execution, each artifact instance is interacting to each other forming the evolution of the system. The GSM information model uses (possibly nested) attribute/value tuples to capture the domain of interest. The key elements of a lifecycle model are *stages*, *milestones* and *guards*. Stages are possibly hierarchical clusters of activities, intended to update and extend the data of the information model. They are associated to milestones, business operational objectives which can be achieved while the stage is under execution. Each stage has one or more guards, which control the activation of stages and, like milestones, are described in terms of data-aware expressions, involving conditions over the artifact information model.

9.5.2 OBGSM System Specification

The OBGSM tool takes a *conceptual temporal property* specified over the Semantic Layer of a GSM-based artifact system, and then producing a corresponding temporal property that can be directly verified by GSMC over the GSM specification, without involving the Semantic Layer anymore. Specifically, OBGSM has three inputs:

1. a conceptual temporal property Φ ;
2. an OWL 2 QL³ TBox;
3. a mapping specification \mathcal{M} .

In the following, we detail the languages used to specify Φ and \mathcal{M} .

² <http://www.omg.org/spec/CMMN/>

³ http://www.w3.org/TR/2008/WD-owl2-profiles-20081008/#OWL_2_QL. OWL 2 QL is the OWL2 profile that closely corresponds to the DL-Lite family of Description Logics.

9.5.2.1 Specification of Conceptual Temporal Properties

For the temporal component of the conceptual temporal properties, we rely on CTL, in accordance to the input verification language of GSMC. Remember that CTL is subsumed by μ -calculus [86, 98]. As far as the local queries over the ontology are concerned, the language relies on a fragment of SPARQL, in accordance to the query language supported by Ontop. More specifically, the syntax of the conceptual temporal properties is as follows:

```

formula ::= [ query ]
          | (formula)
          | formula AND formula
          | formula OR formula
          | formula -> formula
          | ! formula
          | AG formula
          | EG formula
          | AF formula
          | EF formula
          | AX formula
          | EX formula
          | A (formula UNTIL formula)
          | E (formula UNTIL formula)
          | FORALL Var . forallQuantification
          | EXISTS Var . existsQuantification

forallQuantification ::= [ query ] -> formula
                      | FORALL Var . forallQuantification
                      | [ query ]

existsQuantification ::= [ query ] AND formula
                       | EXISTS Var . existsQuantification
                       | [ query ]

```

where [query] is a SPARQL 1.1⁴ SELECT query. SELECT queries, in turn, obey to the following grammar:

```

query ::= PrefixDeclarations SELECT Var WHERE {Triples FILTER(filter)}

filter ::= filterExpression
        | filterExpression && filter

```

⁴ <http://www.w3.org/TR/sparql11-query/>

```

filterExpression ::= var_const < var_const
                  | var_const <= var_const
                  | var_const > var_const
                  | var_const >= var_const
                  | var_const = var_const
                  | var_const != var_const

var_const ::= Var
           | integer
           | "string"
           | "string"^^http://www.w3.org/2001/XMLSchema#string
           | "string"^^http://www.w3.org/2001/XMLSchema#integer
           | "string"^^http://www.w3.org/2001/XMLSchema#decimal
           | "string"^^http://www.w3.org/2001/XMLSchema#double
           | "string"^^http://www.w3.org/2001/XMLSchema#dateTime
           | "string"^^http://www.w3.org/2001/XMLSchema#boolean
           | "string"^^http://www.w3.org/1999/02/22-rdf-syntax-ns#Literal

```

where:

- **Var** is a variable that obeys to the pattern $?([a-z] | [A-Z])^+$;
- **Triples** and **PrefixDeclarations** follow the usual triple patterns and prefix declarations of SPARQL 1.1;
- **integer** and **string** are the standard integer and string built-in domains.

Additionally, we require that all variables present in the **SELECT** clause of the query also appear in the **WHERE** clause, and vice-versa; in other words, all variables in the query must be answer variables.

The semantics of the temporal operators is as in CTL [27]. For the first order quantification, we impose the following restrictions:

- Only closed temporal formulas are supported for verification.
- Each first-order quantifier must be “guarded”, in such a way that it ranges over constants present in the current active domain. This active domain quantification is in line with GSMC, and also with the $\mu\mathcal{L}_A^{\text{EQL}}$ logics. As attested by the grammar above, this is syntactically guaranteed by requiring quantified variables to appear in a [query] according to the following guidelines:

$$\forall \vec{x}. \text{query}(\vec{x}) \rightarrow \phi$$

$$\exists \vec{x}. \text{query}(\vec{x}) \wedge \phi$$

- Quantified variables must obey to specific restrictions, depending on whether they quantify over object terms or values. This can be syntactically recognized by checking whether the variable appears in the second component of an attribute (in this case, it ranges over values) or not. The restriction is as follows: for each variable y ranging over values, there must be at least one variable x that ranges over object terms and that appears in the first component of the corresponding attribute (i.e., $\text{Attr}(x, y)$ is present in the query, with Attr being an attribute of the TBox), such that x is quantified “before” y . For example, $\forall x. C(x) \implies \exists y. \text{Attr}(x, y)$ satisfies this condition, whereas $\exists y \exists x. \text{Attr}(x, y)$ does not.

These restrictions have been introduced so as to guarantee that the conceptual temporal property can be translated into a corresponding GSMC temporal property. In fact, GSMC poses several restrictions on the way values can be accessed.

Example 9.17. We consider a simple university information system in order to provide an example while explaining the system specification of OBGSM. The following TBox is used to capture the relevant concepts and relations of the university domain at the Semantic Layer:

Bachelor \sqsubseteq	Student	$\delta(\text{MNum}) \sqsubseteq$	Student
Master \sqsubseteq	Student	$\delta(\text{HasAge}) \sqsubseteq$	Student
Graduated \sqsubseteq	Student	$\exists \text{Attend} \sqsubseteq$	Student
		$\exists \text{Attend}^- \sqsubseteq$	Course

The Artifact Layer contains the following artifact types:

1. **ENROLLEDSTUDENT**, whose instances represent the enrolled students. For each enrolled student, these data attributes are maintained: **ID**, **MNum**, **Name**, **Age**, **Type**, where **ID**, **Name** and **Type** are of type String, while **Age** and **MNum** are of type Integer.
2. **GRAD**, whose instances represent those students who have been graduated. The following data attributes are maintained: **ID**, **MNum**.
3. **COURSE**, whose instances represent the courses offered by the university. They have the following data attributes: **ID**, **CourseName**.

An example of temporal property specified over the Semantic Layer is:

```
EF FORALL ?x. ([PREFIX : <http://acsi/example/student/student.owl#>
  PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
  SELECT ?x WHERE { ?x rdf:type :Bachelor } ] ->
[PREFIX : <http://acsi/example/student/student.owl#>
  SELECT ?x WHERE { ?x rdf:type :Graduated } ]
);
```

which says that “eventually there is a state in the future where all bachelor students are graduated”. Another example is:

```
EF FORALL ?x. ([PREFIX : <http://acsi/example/student/student.owl#>
  PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
  PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
  SELECT ?x WHERE
    { ?x rdf:type :Master; :HasAge "26"^^xsd:integer } ]
-> [PREFIX : <http://acsi/example/student/student.owl#>
  SELECT ?x WHERE { ?x rdf:type :Graduated } ]
);
```

It states that “eventually in the future there is a state where all bachelor students who are at least 26 years old are graduated”.

Notice that in the second temporal property of Example 9.17, the typed value "26"^^xsd:integer is used to denote the age of students. More in general, according to the current implementation of -ONTOP-, there is support for the following type of values:

- xsd:string
- xsd:integer
- xsd:decimal
- xsd:double
- xsd:dateTime
- xsd:boolean
- rdf:Literal

where “xsd:” and “rdf:” are predefined prefixes, respectively defined as “xsd: http://www.w3.org/2001/XMLSchema#” and “rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#”. Whenever an input value is not typed, we consider it to be, by default, of type `rdf:Literal`.

9.5.2.2 Specification of the Input Mapping

The structure of our mapping language is borrowed from the one of -ONTOP-. More specifically, the expected file format is:

```
[PrefixDeclaration]
...

[ClassDeclaration] @collection [[
...
]]

[ObjectPropertyDeclaration] @collection [[
...
]]

[DataPropertyDeclaration] @collection [[
...
]]

[MappingDeclaration] @collection [[
...
]]
```

In the following, we detail the different parts of this format.

The `[PrefixDeclaration]` part contains the definition of the URI (Uniform Resource Identifier) prefixes that will be used in the remainder of the file.

Example 9.18. We provide a simple prefix declaration that could be contained in a mapping specification file:

```
[PrefixDeclaration]
xsd: http://www.w3.org/2001/XMLSchema#
rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
: http://acsi/example/student/student.owl#
```

Parts [ClassDeclaration], [ObjectPropertyDeclaration], and [DataPropertyDeclaration], respectively contain the declaration of *concepts*, *roles*, and *attributes* name that will be mentioned in the mapping declarations. They are also specified in terms of URIs, where each entry is separated by comma.

Example 9.19. We provide three sample declarations for a class, an object property, and a data property, respectively:

```
[ClassDeclaration] @collection [[
:Student, :Bachelor, :Graduated, :Master, :Course
]]

[ObjectPropertyDeclaration] @collection [[
:Attend
]]

[DataPropertyDeclaration] @collection [[
:MNum, :HasAge
]]
```

The [MappingDeclaration] contains the declaration of mapping assertions (cf. Definition 9.1). When constructing object terms starting from Relational Layer, we require that only unary function symbols are used. As in -ONTOP-, such unary function symbols are in turn represented by *URI templates* (i.e., a pre-set format for URIs). For example, the object term $stud(x)$ is represented as $\langle \text{http://www.acsi-project.eu/example/\#stud\{x\}} \rangle$.

Each mapping assertion is then described by three components:

1. **mappingId**, which provides a unique identifier for the mapping assertion.
2. **target**, which contains the *target query* (i.e., the head of the mapping). Technically, a target query is a CQ over the vocabulary of the ontology. For the specification of such target query, we adopt the -ONTOP- syntax, which is based on the Turtle⁵ syntax to represent RDF triples. Each atom in the CQ is in fact represented as an RDF-like triple template. There are three kinds of possible atoms in the target query:

- a) Concepts, expressed as

⁵ <http://www.w3.org/TR/turtle/>

[URI_Template] rdf:type [ConceptName]

where [ConceptName] is an URI, and rdf: is the prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>. For example, to represent the atom

$$\text{ConceptName}(c(x))$$

(where *ConceptName* is a concept name in the ontology), the following notation is used:

<"&;c{\$x}"> rdf:type :ConceptName

where “:” is a predefined prefix.

b) Roles, again expressed as triples:

[URI_Template] [RoleName] [URI_Template]

where [RoleName] is an URI. For example, the atom

$$\text{RoleName}(r1(x), r2(y))$$

(where *RoleName* is a role name in the ontology) is represented as:

<"&;r1{\$x}"> :RoleName <"&;r2{\$y}">

where “:” is a predefined prefix.

c) Attributes, whose definition resembles the one of roles:

[URI_Template] [AttributeName] [TypedOrUntypedVariable]

where [AttributeName] is an URI. For example,

$$\text{AttributeName}(\text{att}(x), \text{integer}(y))$$

is represented as:

<"&;att{\$x}"> :AttributeName \$y^{~xsd:integer} .

where “:” and “xsd:” are predefined prefixes, and “xsd:” is defined as “xsd: <http://www.w3.org/2001/XMLSchema#>”. It is worth noting that the second component of an attribute is a value. We assume that it originates from a value attribute contained inside the information model of an artifact, we use dedicated function symbols to wrap the value into an object term, ensuring that this choice does not overlap with any function symbol chosen for “real” object terms. In the example above, we use “<http://www.w3.org/2001/XMLSchema#integer>”, but in general, -ONTOP- supports all the following special data types:

- <http://www.w3.org/2001/XMLSchema#string>
- <http://www.w3.org/2001/XMLSchema#integer>
- <http://www.w3.org/2001/XMLSchema#decimal>
- <http://www.w3.org/2001/XMLSchema#double>
- <http://www.w3.org/2001/XMLSchema#dateTime>
- <http://www.w3.org/2001/XMLSchema#boolean>

- <http://www.w3.org/1999/02/22-rdf-syntax-ns#Literal>

3. **source**, which describes the *source query*, i.e., the body of the mapping. The grammar of the source query is borrowed from the grammar of the GSMC input language [113], with extensions that allow to “extract” artifact identifiers and their value attributes, so as to link them to the ontology. The extended syntax is:

```

expression ::= constant
              | expression == ?variable
              | expression aop expression
              | expression lop expression
              | {variable./path/attributeID}
              | GSM.isStageActive('variable','stageID')
              | GSM.isMilestoneAchieved('variable','milestoneID')
              | variable.attributeID1 -> exists(attributeID2 = expression)

formula ::= expression
          | (formula)
          | formula AND formula
          | formula OR formula
          | ! formula
          | exists('variable', 'artifactID')(formula)
          | forall('variable', 'artifactID')(formula)
          | get('variable', 'artifactID')(formula)

```

The two key additional features rely in the possibility of introducing a variable assigning it to an expression (see the second line in the grammar definition), and the possibility of “getting” a variable representing an instance of the specified artifact (see the last line in the grammar definition). These variables are considered to be free in the specified query, and can be consequently used to “transport” values and artifact identifiers into the Semantic Layer, respectively as attributes and object terms.

Example 9.20. Consider again the Artifact and Semantic Layer introduced in Example 9.17. We specify the following mapping assertions to link the three artifacts and their information models to the ontology present in the Semantic Layer:


```

[MappingDeclaration] @collection [[

mappingId    BachelorStudent
target       <"&::stud/{x}/"> rdf:type :Bachelor .
source       get('x','ENROLLEDSTUDENT')
              ({x./ENROLLEDSTUDENT/Type} == "Bachelor")

mappingId    MasterStudent
target       <"&::stud/{x}/"> rdf:type :Master .
source       get('x','ENROLLEDSTUDENT')
              ({x./ENROLLEDSTUDENT/Type} == "Master")

mappingId    MatriculationNumber
target       <"&::stud/{x}/"> :MNum $y^^xsd:integer .
source       get('x','ENROLLEDSTUDENT')({x./ENROLLEDSTUDENT/MNum} ==
?y)

mappingId    GraduatedStudent
target       <"&::stud/{x}/"> rdf:type :Graduated .
source       get('x','ENROLLEDSTUDENT')(exists('y', 'GRAD')(
{x./ENROLLEDSTUDENT/MNum} == {y./GRAD/MNum}))

mappingId    Age
target       <"&::stud/{x}/"> :HasAge $y^^xsd:integer .
source       get('x','ENROLLEDSTUDENT')(x./ENROLLEDSTUDENT/Age == ?y)

mappingId    AttendingCourse
target       <"&::stud/{x}/"> :Attend <"&::course/{y}/"> .
source       get('x','ENROLLEDSTUDENT')(get('y','COURSE')
(x./ENROLLEDSTUDENT/AttendedCourses->exists(
ID == {y./COURSE/ID})))

]]

```

The first two mapping assertions are used to populate bachelor and master students in the Semantic Layer, by extracting information from artifact instances of type `ENROLLEDSTUDENT`, respectively selecting those instances whose `Type` field corresponds to the string “Bachelor” or “Master”. Notice that the artifact instance identifier x is used to create the corresponding student object term $stud(x)$ in the ontology.

The following three mapping assertions are used to populate attributes in the Semantic Layer, starting from specific artifacts and fields in their information models. According to the previously discussed restrictions, the first component of attributes is always associated to an object term constructed starting from an artifact instance identifier, and the second from a value in its information model.

The last mapping assertions is used to populate a relation in the Semantic Layer, starting from pairs of artifact identifiers in the Artifact Layer. In particular, the source query is used to extract all pairs of artifact instances of type `ENROLLEDSTUDENT` and `COURSE`, such that the course artifact instance is among the attended courses by the student artifact instance (notice the navigation `x./ENROLLEDSTUDENT/AttendedCourses` to select all attended courses, and the conse-

quent join used to check whether the considered course instance is among the attended ones). In this case, both the first and the second component of the association are object terms constructed from artifact instance identifiers.

9.5.2.3 OBGSM Workflow and Components

As depicted in the Figure 11, the workflow of OBGSM is as follows.

1. The tool reads and parses the input conceptual temporal property Φ , the input ontology (TBox) T , and the input mapping declaration \mathcal{M} .
2. The tool *rewrites* the input conceptual temporal property Φ based on the input ontology (TBox) T , in order to compile away the TBox. This step produces *rewritten temporal property* $rew(\Phi, T)$.
3. The rewritten property $rew(\Phi, T)$ is *unfolded* by exploiting \mathcal{M} . The final temporal property $\Phi_{GSM} = \text{UNFOLD}(rew(\Phi, T), \mathcal{M})$ obeys to the syntax expected by GSMC, and is such that verifying Φ over the transition system of the GSM model under study after projecting its states into the Semantic Layer through \mathcal{M} , is equivalent to verifying Φ_{GSM} directly over the GSM model (without considering the Semantic Layer).
4. GSMC is invoked by passing Φ_{GSM} together with the specification file of the GSM model under study.

Notice that the correctness of the translation is guaranteed by the fact OBGSM manipulates the local components of the query Φ according to the standard rewriting and unfolding algorithms, while maintaining untouched the temporal structure of the property. This has been proven to be the correct way of manipulating the temporal property as in SEDAPs (cf. Section 9.4). The proof has been done for $\mu\mathcal{L}_A^{\text{EQL}}$, and since first-order CTL with active domain quantification is a fragment of $\mu\mathcal{L}_A^{\text{EQL}}$, the result directly applies also in our setting. This result also shows that OBGSM can largely rely on state-of-the-art existing rewriting and unfolding techniques to manipulate the temporal properties. Indeed, OBGSM exploits -ONTOP- to accomplish this task, by also adding a last step to deal with the constructs that have been introduced for the mapping assertions, but that are not directly supported by GSMC. In particular, OBGSM turn “get” statements into corresponding the quantifications.

OBGSM consists of the following main components (see also Figure 11):

1. *Temporal Property Parser and Validator*. This component parses and validates a conceptual temporal property, checking its well-formedness and whether it

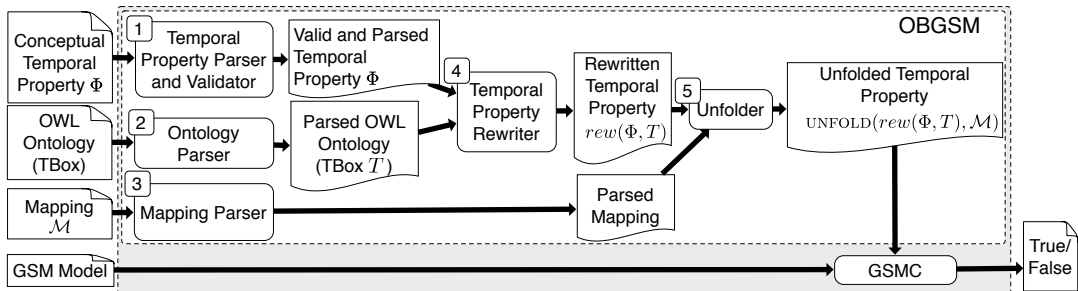


Figure 11: OBGSM System Architecture

guarantees the required restrictions or not. The parser for the temporal part of the property is implemented using Antlr 4.0⁶. For parsing the local queries in the temporal property, the SPARQL 1.1 parser component from -ONTOP- is extensively used, together with the Apache JenaTM library⁷.

2. *Ontology Parser*. This component, entirely provided by -ONTOP-, reads and parse an OWL 2 QL ontology.
3. *Mapping Parser*. This component reads and parses the file containing mapping assertions. As for the implementation, the parser already present in -ONTOP- is reused and complemented with the additional features by using Antlr 4.0.
4. *Temporal Property Rewriter*. When the input temporal property and the input ontology have been parsed, OBGSM *rewrites* the parsed temporal property based on the given ontology, using this component. The implementation of the query rewriting functionality is fully inherited from -ONTOP-.
5. *Unfolder*. This component takes the specification of mapping assertions and the property produced by the rewriter, producing the final unfolded property. To do so, it extends the base unfolding functionality already present in -ONTOP-.

9.5.3 A Use Case Example

As a case study to demonstrate our approach we refer to the fragment of the Energy Use Case Scenario developed within the ACSI Project [173, 174, 175, 176]. We show how the Semantic Layer can be exploited in order to facilitate the specification of temporal properties of interest, and discuss how these are automatically translated into properties that can be directly verified by GSMC over the Energy GSM model.

9.5.3.1 ACSI Energy Use Case at a Glance

We sketch here the main aspects of the Energy use case, and referring the interested reader to [173, 174, 175, 176] for further details.

The ACSI Energy use case focuses on the electricity supply exchange process between electric companies inside a distribution network. The electricity exchange between companies occurs at *control points*. Within a control point, a measurement of electricity supply exchange takes place in order to calculate the fair remuneration that the participating companies in the control point should receive. The measurement is done by a *meter reader company*, which corresponds to one of the companies pertaining to that particular certain control point. The measurement results from the control points are then submitted to the *system operator*, who is in charge of processing the results and publishing a *control point monthly report*. A participating company can raise an objection concerning the published measurement. Once all the risen objections are resolved, the report is closed. The collection of CP monthly reports is then represented as a *CP monthly report accumulator*.

⁶ <http://wwwantlr.org>

⁷ <http://jena.apache.org>

9.5.3.2 The GSM Model for ACSI Energy Use Case

The sketched ACSI energy use case scenario is implemented by considering two artifacts:

- *Control Point Monthly Report (CPMR)*. This artifact contains the information about hourly measurements done in a control point within a certain month. The lifecycle of an instance of this artifact is started when the Metering Data Management (MDM) system provides the hourly measurements, and runs until the liquidation for the CP measurements is started. This artifact consists of three root stages:
 - *CPMRInitialization*, activated when a new instance of the CPMR artifact is created.
 - *Claiming*, This stage handles the submission of measurements, and the consequent reviewing stage, where objections may be raised. Five sub-stages are used to deal with this lifecycle in a fine-grained way: *Drafting*, *Evaluating*, *Reviewing*, *Closing*, and *CreateObjection*.
 - *MeasurementUpdating*, activated when there is an event requesting for updating the measurement results.
- *CPMR Monthly Accumulator (CPMRMA)*, responsible for the measurement files. It submits measurements to the system operator, and receives back from the operator the value for the corresponding official measurements. It consists of four root stages:
 - *CPMRMAInitialization*, which handles the generation of measurement files.
 - *SubmittingMeasurementFile*, which submits the measurement files to the system operator.
 - *EvaluatingMeasurementFile*, which waits the official measurements from the system operator.
 - *ProcessingOfficialMeasurement*, which calculates the differences between the official measurements and the submitted measurements, and consequently notify the CPMR instances about the differences. The *CalculatingDifferences* and *NotifyingOfficialMeasurement* sub-stages are used to handle this portion of the lifecycle.

Figure 12 shows the GSM model for the two artifacts described above.

9.5.3.3 The Semantic Layer Specification

We provide a Semantic Layer on top of the GSM model for the ACSI Energy use case, restricting our attention to the Published Control Point Measurement Report (CPMR).

9.5.3.4 The Ontology

A UML model for the ontology of the Semantic Layer is depicted in Figure 13. A control point measurement report can be either:

- a finished CPMR (when the milestone *CPMRFinished* is achieved),

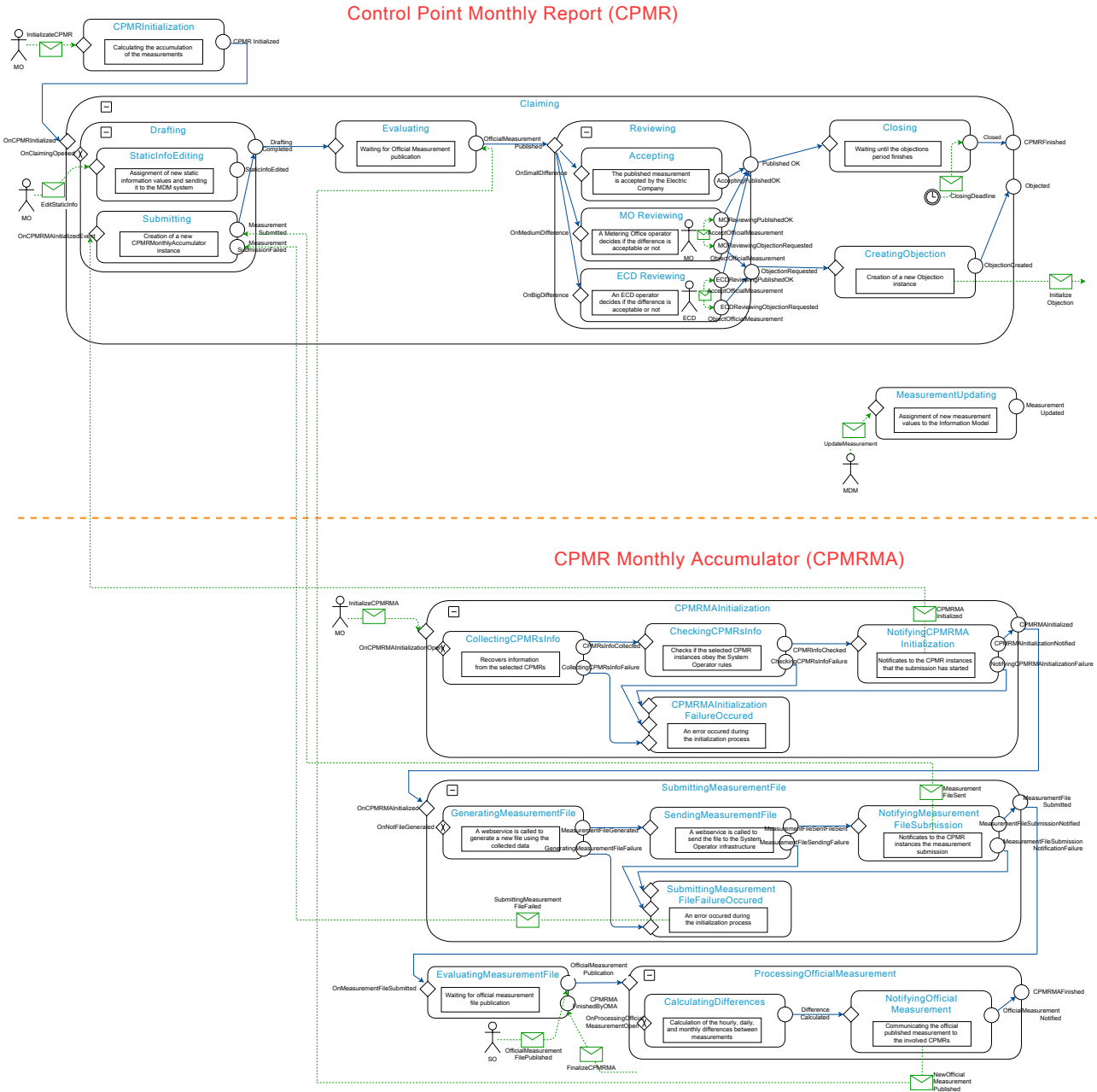


Figure 12: GSM Model for ACSI Energy Use Case (The picture is from [60])

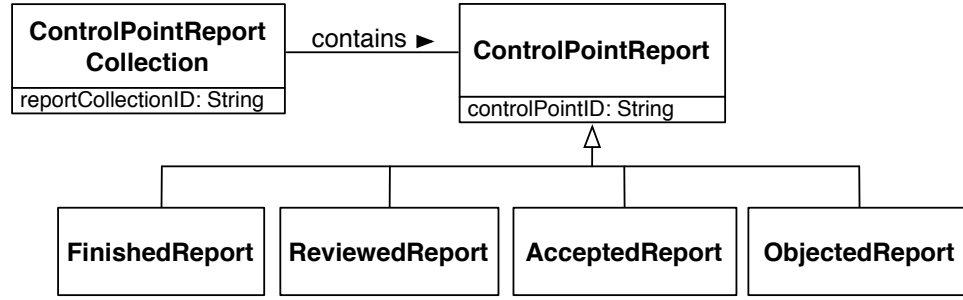


Figure 13: Ontology for the CPMR reviewing process in ACSI Energy Use Case

- a reviewed CPMR (after finishing the review inside the *Reviewing* stage),
- an accepted CPMR (when the milestone *PublishedOK* is achieved),
- an objected CPMR.

We formalize the UML model in $DL-Lite_A$:

$$\begin{aligned}
 \text{FinishedReport} &\sqsubseteq \text{ControlPointReport} \\
 \text{ReviewedReport} &\sqsubseteq \text{ControlPointReport} \\
 \text{AcceptedReport} &\sqsubseteq \text{ControlPointReport} \\
 \text{ObjectedReport} &\sqsubseteq \text{ControlPointReport} \\
 \exists \text{contains} &\sqsubseteq \text{ControlPointReportCollection} \\
 \exists \text{contains}^- &\sqsubseteq \text{ControlPointReport} \\
 \delta(\text{controlPointID}) &\sqsubseteq \text{ControlPointReport} \\
 \rho(\text{controlPointID}) &\sqsubseteq \text{String}
 \end{aligned}$$

This ontology is shown visually in Figure 13.

9.5.3.5 The Mapping Assertions

We use the following mapping assertions in order to link the information model in the GSM model to the Semantic Layer. The assertions are written in the mapping specification language of OBGSM as follows.

```

[PrefixDeclaration]
xsd: http://www.w3.org/2001/XMLSchema#
owl: http://www.w3.org/2002/07/owl#
: http://acsi/example/ACSIEnergy/ACSIEnergy.owl#
rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#

[ClassDeclaration] @collection [[
  owl:Thing, :ControlPointReportCollection, :ControlPointReport,
  :ObjectedReport, :AcceptedReport, :ReviewedReport, :FinishedReport
]]

[ObjectPropertyDeclaration] @collection [[
  :contains
]]
  
```

```

[DataPropertyDeclaration] @collection [[
  :hasControlPointID
]]

[MappingDeclaration] @collection [[
  mappingId ControlPointReportCollectionMapping
  target    <"&::cpmrma/{x}/"> rdf:type :ControlPointReportCollection .
  source    get('x', 'CPMRMA')(TRUE)

  mappingId ControlPointIDMapping
  target    <"&::cpmr/{x}/"> :hasControlPointID $y~xsd:string .
  source    get('x', 'CPMR')({x./CPMR/CPID} == ?y)

  mappingId CPMRMAContainsCPMRMapping
  target    <"&::cpmrma/{x}/"> :contains <"&::cpmr/{y}/"> .
  source    get('x', 'CPMRMA')(get('y', 'CPMR')(
    x./CPMR/CPMRDATA->exists(CPMRID == {y./CPMR/ID})))

  mappingId ReviewedReportMapping
  target    <"&::cpmr/{x}/"> rdf:type :ReviewedReport .
  source    get('x', 'CPMR')(
    GSM.isMilestoneAchieved('x', 'AcceptingPublishedOK') OR
    GSM.isMilestoneAchieved('x', 'MOReviewingPublishedOK') OR
    GSM.isMilestoneAchieved('x', 'ECDReviewingPublishedOK'))

  mappingId AcceptedReportMapping
  target    <"&::cpmr/{x}/"> rdf:type :AcceptedReport .
  source    get('x', 'CPMR')(GSM.isMilestoneAchieved('x', 'PublishedOK'))

  mappingId ObjectedReportMapping
  target    <"&::cpmr/{x}/"> rdf:type :ObjectedReport .
  source    get('x', 'CPMR')(GSM.isMilestoneAchieved('x', 'Objected') OR
    GSM.isMilestoneAchieved('x', 'ObjectionRequested') OR
    GSM.isMilestoneAchieved('x', 'ObjectionCreated') OR
    GSM.isMilestoneAchieved('x', 'MOReviewingObjectionRequested') OR
    GSM.isMilestoneAchieved('x', 'ECDReviewingObjectionRequested'))

  mappingId FinishedReportMapping
  target    <"&::cpmr/{x}/"> rdf:type :FinishedReport .
  source    get('x', 'CPMR')(GSM.isMilestoneAchieved('x', 'CPMRFinished'))
]]

```

Where “:” is a prefix declared as <http://acsi/example/energy/energy.owl#>. The intuition of some mapping assertions above is as follows:

- `ControlPointReportMapping` and `ControlPointReportCollectionMapping` populate the concepts *ControlPointReport* and *ControlPointReportCollection* with the CPMR and CPMRMA artifact instances respectively.
- `CPMRMAContainsCPMRMapping` populates the *contains* role, which relates CPMRMA with the CPMRs it contains.

- **ControlPointIDMapping** populates the attribute *hasControlPointID* by relating *ControlPointReport* with its control point ID.
- **ReviewedReportMapping** populates the *ReviewedReport* concept with the CPMR artifact instance, given the achievement of one of the three milestones *AcceptingPublishedOK*, *MOReviewingPublishedOK* or *ECDReviewingPublishedOK*. In the Semantic Layer, *ReviewedReport* intuitively represents a CPMR that has been reviewed. In the Artifact Layer, this corresponds to the situation in which the CPMR has been reviewed and accepted either by the electric company, or by the metering office (MO), or by the Electric Control Department (ECD). This example show how such details can be hidden from the Semantic Layer, which does not show the fact that a *ReviewedReport* is obtained by a (possibly complex) chaining of achieved milestones in the underlying GSM model.
- **AcceptedReportMapping** populates the *AcceptedReport* concept with the CPMR artifact instance, when milestone *PublishedOK* is achieved. This intuitively means that the *AcceptedReport* is a published CPMR that has been approved.
- **ObjectedReportMapping** populates the *ObjectedReport* concept with an objected CPMR artifact instance. This situation is recognized, at the Artifact Layer, by combining different related milestones.
- **FinishedReportMapping** populates the *FinishedReport* concept with the finished CPMRs, i.e., those that have achieved milestone *CPMRFinished*.

9.5.3.6 Verification

In this section, we demonstrate how the presence of the Semantic Layer can help in the specification of temporal properties of interests. We consider in particular the following properties:

1. All control point reports will eventually will be finished.
2. All control point reports that are accepted must have been reviewed. This property is used to ensure that there is no way to achieve a state in which a certain CPMR is accepted, without going through the review for that CPMR. Notice that “having being reviewed” is considered to be a permanent property of CPMRs.
3. All control point reports that are finished must not be objected control point reports. This property ensures that control point reports cannot be classified as finished as long as they are still objected.
4. All objected control point reports must not be finished control point reports.

Such four properties can be expressed as conceptual temporal properties over the Semantic Layer. In particular, we encode them using the language provided by OBGSM as follows (Note: for compactness of the presentation, in the following we do not write the queries in SPARQL):

1. $AG(\text{FORALL } x . ([\text{ControlPointReport}(x)] \rightarrow EF[\text{FinishedReport}(x)]))$
2. $AG(\text{FORALL } x . ([\text{AcceptedReport}(x)] \rightarrow [\text{ReviewedReport}(x)]))$
3. $AG(\text{FORALL } x . ([\text{FinishedReport}(x)] \rightarrow ![\text{ObjectedReport}(x)]))$
4. $AG(\text{FORALL } x . ([\text{ObjectedReport}(x)] \rightarrow ![\text{FinishedReport}(x)]))$

We now show how this high-level property are compiled by OBGSM into underlying temporal properties that can be fed into the GSMC model checker. We stress that,

without the presence of the Semantic Layer, the user would be forced to write this low-level properties manually.

1. The rewriting step for the first conceptual temporal property produces the following formula, which “embeds” the constraints of the ontology present at the Semantic Layer:

$$\begin{aligned} \text{AG}(\text{FORALL } x . ((\text{EXISTS } y . [\text{hasControlPointID}(x, y)]) \text{ OR} \\ [\text{ObjectedReport}(x)] \text{ OR} \\ [\text{AcceptedReport}(x)] \text{ OR} \\ [\text{ControlPointReport}(x)] \text{ OR} \\ [\text{ReviewedReport}(x)] \text{ OR} \\ (\text{EXISTS } z . [\text{contains}(z, x)]) \text{ OR} \\ [\text{FinishedReport}(x)] \text{ OR} \\ \rightarrow \text{EF}[\text{FinishedReport}(x)])) \end{aligned}$$

The expansion of the queries contained in the property is done by embedding the following cases, reflected by the ontology constraints:

- Those objects that have a control point ID (i.e., are in the domain of the attribute `hasControlPointID`), are instances of `ControlPointReport`.
- `ObjectedReport` is a `ControlPointReport`.
- `AcceptedReport` is a `ControlPointReport`.
- `ReviewedReport` is a `ControlPointReport`.
- Those objects that are in the range of the role `contains` are instances of `ControlPointReport`.
- `FinishedReport` is a `ControlPointReport`.

By exploiting the mapping assertions, OBGSM unfolds the rewritten property into this final result:

$$\begin{aligned} \text{AG}(\text{forall}('x', 'CPMR')(!(\text{GSM.isMilestoneAchieved}('x', 'Objected') \text{ OR} \\ \text{GSM.isMilestoneAchieved}('x', 'ObjectionRequested') \text{ OR} \\ \text{GSM.isMilestoneAchieved}('x', 'ObjectionCreated') \text{ OR} \\ \text{GSM.isMilestoneAchieved}('x', 'MOReviewingObjectionRequested') \text{ OR} \\ \text{GSM.isMilestoneAchieved}('x', 'ECDReviewingObjectionRequested') \text{ OR} \\ \text{GSM.isMilestoneAchieved}('x', 'PublishedOK') \text{ OR} \\ (false) \text{ OR} \\ \text{GSM.isMilestoneAchieved}('x', 'AcceptingPublishedOK') \text{ OR} \\ \text{GSM.isMilestoneAchieved}('x', 'MOReviewingPublishedOK') \text{ OR} \\ \text{GSM.isMilestoneAchieved}('x', 'ECDReviewingPublishedOK') \text{ OR} \\ \text{exists}('y', 'CPMRMA')(y./CPMRA/CPMRDATA \rightarrow \\ \text{exists}(CPMRID == x./CPMR/ID)) \text{ OR} \\ \text{GSM.isMilestoneAchieved}('x', 'CPMRFinished') \text{ OR} \\ \text{EF}(\text{GSM.isMilestoneAchieved}('x', 'CPMRFinished')))) \end{aligned}$$

Notice that the absence of a mapping assertion for the concept `ControlPointReport` results into a *false* disjunct in the unfolding.

2. The translation of the second conceptual temporal property produces this final result:

AG **forall**('x', 'CPMR')(!(*GSM.isMilestoneAchieved*('x', 'PublishedOK')) OR (*GSM.isMilestoneAchieved*('x', 'AcceptingPublishedOK') OR *GSM.isMilestoneAchieved*('x', 'MOReviewingPublishedOK') OR *GSM.isMilestoneAchieved*('x', 'ECDReviewingPublishedOK')))

3. The translation of the third conceptual temporal property produces this final result:

AG **forall**('x', 'CPMR')(!*GSM.isMilestoneAchieved*('x', 'CPMRFinished') OR !(*GSM.isMilestoneAchieved*('x', 'Objected') OR *GSM.isMilestoneAchieved*('x', 'ObjectionRequested') OR *GSM.isMilestoneAchieved*('x', 'ObjectionCreated') OR *GSM.isMilestoneAchieved*('x', 'MOReviewingObjectionRequested') OR *GSM.isMilestoneAchieved*('x', 'ECDReviewingObjectionRequested')))

4. The translation of the fourth conceptual temporal property produces this final result:

AG **forall**('x', 'CPMR')(!(*GSM.isMilestoneAchieved*('x', 'Objected') OR *GSM.isMilestoneAchieved*('x', 'ObjectionRequested') OR *GSM.isMilestoneAchieved*('x', 'ObjectionCreated') OR *GSM.isMilestoneAchieved*('x', 'MOReviewingObjectionRequested') OR *GSM.isMilestoneAchieved*('x', 'ECDReviewingObjectionRequested')) OR !*GSM.isMilestoneAchieved*('x', 'CPMRFinished'))

By comparing the properties specified over the Semantic Layer and their corresponding translations, it is apparent that, even in this simple case study, the presence of the Semantic Layer hides low-level details, helps the modeler in focusing on the domain under study, and allows for using the vocabulary he/she is familiar with (i.e., the vocabulary of the ontologies).

CONCLUSION

We close this thesis by recapping our journey so far, and also discussing several plausible future directions.

10.1 Summary

Within this thesis, we have proposed several frameworks for specifying semantically-rich data-aware business processes systems that also take into account various aspects. For each setting, we have addressed the problem of verifying temporal properties over the evolution of the system.

Specifically, in Chapter 4, we introduced a framework for specifying semantically-rich data-aware processes systems, namely Golog-KABs (GKABs), by leveraging on Knowledge and Action Bases (KABs) [121]. Fundamentally, GKABs capture the manipulation of Knowledge Bases (KBs) by the Golog program [134]. We have also introduced standard execution semantics for GKABs that do nothing regarding inconsistency (i.e., updates that lead to an inconsistent state are simply rejected). A GKAB with standard execution semantics is called S-GKAB. Furthermore, we have shown that verification of rich temporal properties over S-GKABs can be reduced to corresponding verification of KABs and vice versa.

In Chapter 5, we extended GKABs towards Inconsistency-aware GKABs (I-GKABs) by incorporating several inconsistency management approaches (based on the notion of repairs). Concerning about verification of I-GKABs, we have proven that they can be reduced to corresponding verification of S-GKABs and vice versa.

Next, in Chapter 6, we proposed Context-Sensitive GKABs (CSGKABs), which are an extension of GKABs that takes into account contextual information. In Chapter 6, we only focused on S-CSGKABs that is CSGKABs with standard execution semantics (i.e., do nothing w.r.t. inconsistency). We have proved that verification of sophisticated context-sensitive temporal properties over S-CSGKABs is reducible to corresponding verification of S-GKABs and vice versa.

In Chapter 7, we introduced an extension of GKABs that takes into account contextual information as well as employs a sophisticated inconsistency handling mechanism. Additionally, we have shown that verification of such an extension can be reduced to verification of S-GKABs and vice versa. As a deeper investigation, in Chapter 8 we proposed another extension, namely Alternating GKABs (AGKABs), that does not only consider contextual information and employ a sophisticated inconsistency handling mechanism, but also exposes each source of non-determinisms (e.g., the choice of action) within a single evolution step. Essentially, AGKABs allow us to have a more fine-grained understanding over the system evolution. We studied verification of temporal properties over AGKABs and showed that it can be reduced to verification of S-GKABs. In addition, the other direction also has been shown.

Prominently, all of our reductions from various GKABs into KABs preserve run-boundedness, which is a restriction that guarantees the decidability of KABs verification.

As an orthogonal approach for specifying semantically-rich data-aware processes systems, in Chapter 9 we proposed a novel framework called Semantically-Enhanced Data-Aware Processes (SEDAPs) that provides a high-level conceptual view over the evolution of a data-aware processes system, by making use of ontologies. We have successfully addressed the problem of verification of temporal properties expressed over the conceptual level in SEDAPs, by showing that the verification can be reduced to the corresponding verification over DCDSs. Not only theoretical results, we have also concretized the concept of SEDAPs into an implementation of a tool.

10.2 Discussion and Related Works

Concerning the restriction to get the decidability of verification, apart from the run-boundedness condition that we have borrowed from [24], the work in [24] also introduces another restriction called *state boundedness*. Essentially, state boundedness constraints the system by requiring that the number of constants in each state of the system is bounded by a generic bound b . Clearly, run boundedness is more restrictive than state boundedness. I.e., run-boundedness implies state boundedness (consider that the bound on the number of constants in each run also bounds the number of constants in each state of the system).

In [91], the authors propose so called *bounded action theories*, which are situation calculus basic action theories [155] in which in each situation, the number of domain objects that belong to any fluent is bounded (i.e., the number of ground fluent atoms in each situation is bounded). Such restriction on the action theories is similar to the notion of state boundedness, but in the setting of situation calculus. The work by [91] studies the verification of expressive temporal properties based on μ -calculus over dynamic systems formalized in bounded action theories, and shows that such problem is decidable. Furthermore [91] also explores several ways to obtain bounded action theories such as (i) blocking the execution of actions that would destroy the bound; (ii) requiring that for each action and situation, the number of tuples that are added to the fluent is less than or equal to those that are deleted; (iii) axiomatizing the notion of *fading fluents* that basically enforce each fluent to eventually become false if it is not (re)-added for some period of time. The work in [110] studies a framework for modeling and verifying agents expressed in situation calculus bounded action theories [91]. Moreover, [110] shows the decidability of temporal properties verification over online executions of the agent, i.e., those executions resulting from the actions that can be performed by the agent. For specifying the temporal properties, such work considers expressive temporal properties based on μ -calculus.

The works in [30, 32] propose data-aware multi-agent systems called *Artifact-Centric Multi Agent Systems* (AC-MASs) and study the verification of a first order variant of CTL in such a setting. In brief, AC-MASs capture the combined behavior of agents in which each agent has a database to maintain its internal data and can perform actions that manipulate the data. To get decidability of verification, those works assume, in addition to *state-boundedness*, also *uniformity*. The notion of uniformity

is actually borrowed from the notion of *genericity* in databases [1], which basically says that *a query is generic* if it is insensitive w.r.t. renaming of constants. In the setting of “relational transition systems” (i.e., transition systems where each state contains relational data and each transition represents an action execution), roughly speaking, the notion of uniformity says that, if we can go from state s to state t by executing an action $\alpha(\vec{x})$ with parameters \vec{p} (i.e., \vec{x} is instantiated with \vec{p}), then if we have a bijection h that renames the constants \vec{c} into \vec{c}' , we have that we can go from state s' to state t' by executing the action $\alpha(\vec{x})$ with parameters \vec{p}' , where s', t' and \vec{p}' are obtained by applying h respectively to s, t and \vec{p} (i.e., renaming according to h , all constants therein). Concerning uniformity, DCDSs, KABs and our various GKABs variants satisfy uniformity as a consequence of the definition of their execution semantics (see [63, 71]). Still related to the boundedness condition, the work by [33] introduce so called Open Multi-Agent Systems (OMAS), i.e., a framework for modeling multi-agent systems in which the agents may enter and leave the system at run time. Moreover, [33] studies the verification of temporal properties over such setting, in which the temporal properties are specified in first order variant of CTL. The authors obtained decidability of verification by restricting that in each state of the system, the number of constants and the agents are bounded (which is similar to state-boundedness). In addition to the works above, also the following works rely on the state-boundedness assumption to obtain decidability result: [111, 63, 71, 146].

As a remark on state boundedness, although our work in this thesis assumes run-boundedness to get decidability, it can be shown that our results carry over even if we adopt state boundedness. The core intuition is that our translations from the various kinds of GKABs into KABs do not introduce an unbounded number of constants (i.e., we only make use of finitely many additional constants).

Concerning our translation that basically “unfold” the given Golog program into a set of condition-action rules and a set of actions in Section 4.4.2, it is worth to mention that there are also some related works on “unfolding” Golog programs into another formalisms with a particular purpose. In [84], the authors provide a systematic mechanism to unfold Golog program into the so called *characteristic graphs* that aim to capture all possible evolution of program (i.e., encode all reachable program configurations). Technically, each vertex in the graph captures a reachable program configurations that denote the remaining program to be executed, and each edge in the graph capture some informations that is related to the changes of remaining program from one vertex to another vertex (e.g., the action that is executed and causes the changes). The work in [106] presents an algorithm to compile arbitrary Golog programs into basic action theories in situation calculus. Such compilation also involves a transformation from Golog program into petri nets. Furthermore, the authors of [28] “unfold” Golog programs into finite state automata.

The works by [14, 15] introduce a DL-based action formalism. Moreover, building on [14, 15], the works [13, 186] study Golog programs [134] in which the atomic actions are formalized as DL-based actions. Within this setting, [13, 186] tackle the problem of verifying LTL-based temporal properties over the execution of Golog program with respect to the given DL KB. In contrast to our work and also the works in [22, 121, 146], the works of [14, 15, 13, 186] consider the semantics of DL-based actions to be specified in terms of manipulation of DL interpretations. Whereas, in our setting, we have that the actions manipulate the data within the DL KB. Therefore, in our

transition systems, we have that each state is labeled by KB instead of DL-models as in [14, 15, 13, 186].

The combination of DLs with temporal logics has been studied extensively within the line of research called Temporal Description Logics (TDLs) [138, 8, 183, 17, 10, 9]. Although they do not have actions that progress the knowledge base over time as in our setting, the augmented temporal operators in TDL describe the dynamic aspects of the knowledge base. Such combinations between DL and temporal logic are based on a two-dimensional semantics, where one dimension is for time and the other dimension is for the DL domain.

10.3 Future Works

We elaborate several plausible future directions of this research as follows:

- **Inconsistency-aware GKABs based on Consistent Query Answering.** Within this thesis, the ASK operation in GKABs corresponds to certain answers computation, and we have approached the problem of inconsistency handling in GKABs, by resorting to an approach based on ABox repairs. This is achieved through our GKABs parametric execution semantics that intuitively allows us to parameterize the TELL operation via defining various filters. An orthogonal approach to the one taken is to maintain ABoxes that are inconsistent with the TBox as states of the transition system, and rely, both for the progression mechanism and for answering queries used in verification, on consistent query answering [35, 132]. Concerning progression mechanism in GKABs, we can accommodate this changing easily by modifying the ASK operation such that it corresponds to consistent query answering computation. Then, we need to redefine the semantics for query using the consistent query answering semantics. The next question is how should we deal with the verification? A plausible approach to answer this question is exploring whether we can emulate the computation of consistent query answer as a Golog program. Then we can try to reduce the verification problem into the verification of S-GKABs that mimics this GKABs extension. Furthermore, it is also challenging to investigate the correspondence between the consistent query answering based approach and the repair-based approach in dealing with inconsistency in GKABs.
- **Adopting Repair-based Semantics to SEDAPs.** In a SEDAP, an action execution that leads to inconsistency in the semantic layer is rejected. Hence, we reject inconsistent states. However, the inconsistency in a state might be caused by only a small portion of the ABox in the semantic layer and the other consistent portion might be still useful for some reasoning. Therefore, keeping the state by removing the small portion of ABox that made the state inconsistent might be useful. A possible solution to this situation is to “repair” the ABox (in the semantic layer) and allow the action that was rejected to be executed together with some repair in order to maintain the consistency. Notice that the repair happens in the semantic layer and it might lead to several possible repaired states. One question is how we will do the repair. In addition, one needs to understand how the repair in the semantic layer will affect the relational

layer, i.e., the question is how we propagate down the repaired ABoxes in the semantic layer into the corresponding database instances in the relational layer. This problem might have some connections with the problem of view updates [29, 88, 116], because we can consider the semantic layer as a view of the relational layer. To formalize the relation between the two layers, one might also look into bidirectional transformations (c.f. [150]). Also an interesting task to investigate is the verification of conceptual temporal properties over this setting.

- **Embracing Context into SEDAPs.** A further challenging research direction that can be pursued immediately is about adding the notion of context to SEDAP. The idea is as follows: In SEDAP, the intensional knowledge about the domain (which is expressed as a *DL-Lite_A* TBox) is fixed along the evolution of the system. However, this situation is in general too restrictive, since specific knowledge might hold or be applicable only in specific *context-dependent* circumstances. Ideally, one should be able to express the knowledge that is known to be true in certain cases, but not necessarily in all. Having gained the understanding of how to integrate contextual information into GKABs, we are now investigating how we could extend this result to the SEDAP setting. Within SEDAP, we need to understand how to introduce the contextual information over the setting as well as how the contextual information affects the system evolution. Moreover, notice that the verification of SEDAP relies on the notion of “rewriting” and “unfolding”, which takes into account the mapping and relies on the assumption that the TBox is fixed. Given that in the presence of contextual information the TBox is changing over the time with the context, the question is how we should “rewrite” the given conceptual properties. Also, which TBox should we use? A promising route towards tackling this issue is by: (i) Compiling the given query into a disjunction of all possible rewritten queries based on all possible contexts. (ii) Additionally, we conjunct each rewritten query with a query that represents the corresponding context. Essentially, this approach is similar to the way of how we introduce contextually compiled query in Definition 6.34.
- **Inconsistency-Aware Context-Sensitive SEDAP.** Following the last two future works, i.e., adopting repair-based semantics to SEDAPs as well as incorporating context into SEDAPs, it is natural to proceed further by combining those directions towards Inconsistency-Aware Context-Sensitive SEDAP. The presence of context that indirectly changes the TBox requires us to adapt the repair mechanism based on the context changes since we need to do the repair based on the TBox. In addition, the setting of SEDAP, that separates the semantic and relational layer, also complicates the problem.
- **Accommodating Update on Semantic Layer.** Within the setting of SEDAPs, another interesting direction is to investigate the situation where there is a change explicitly over the ontologies in the semantic layer (in the conceptual level). For example when there is a new process introduced in the semantic layer. In this case, the new introduced process might change a certain ABox in the semantic layer. The question here is how such a change will/can affect the relational layer.

- **Process Synthesis.** In the SEDAPs, the processes are specified over the relational layer, and we have a conceptual view provided by the semantic layer which is obtained by projecting the evolution happening in the relational layer to the semantic layer by using the mapping. One interesting alternative setting is to consider the situation where the processes are specified over the semantic layer. In other words, in this setting we have a relational database in the relational layer, and the processes are specified in a high-level way through the semantic layer. Then, we are interested on how this high-level process specification can be “brought down” into the relational layer and executed in the relational layer as well as evolve the existing database. This leads to the problem of synthesizing the process in the relational layer from the given high level specification in the semantic layer, which might also be expected to satisfy some temporal properties specified over the semantic layer. Still about synthesis, it is also interesting to investigate a setting where the process is partly specified and then we synthesize additional process components such that the system satisfies some set of specified temporal properties.
- **Semantic Compliance Checking.** Another interesting setting related to SEDAPs is to consider the situation where we have the process specification on both semantic and relational layer. Then it is interesting to see how the process specifications in these two layers are matched and how they evolve the data as well as satisfying the specified temporal properties. In this scenario, the interesting task is to check if the evolution of our actual/concrete system complies with a certain “evolution requirement”. In this setting, the processes specified in the relational layer can be considered as our actual/concrete system and the processes specified in the semantic layer can be considered as the “evolution requirement” to be checked. Hence in this case we are interested to check if the evolution in the relational layer matches the evolution in the semantic layer. We call the problem “Semantic Compliance Checking” because intuitively we can think that the system evolution happened in the semantic layer is the semantic of the system evolution.
- **Bringing Theory to Practice.** an obvious future direction of this work is of course to implement the theoretical results that has been obtained. Since it has been shown that all variants of our proposed frameworks can be reduced to DCDSs, one possible direction is to take advantage from the works that attempt to implement DCDSs (e.g., [163, 73, 74]). I.e., we could simply implement a translation to transform our systems into DCDSs. Furthermore, there is also a recent work provided by [78] that studies the planning problem in the setting of KABs. Specifically, [78] studies the problem of plan existence over KABs as well as the problem of plan synthesis. Not only providing theoretical results, [78] also implements the results (by also employing off-the-shelf planner system). Essentially, the planning over KABs addresses the problem of checking whether starting from the initial state of a KAB, we can have a sequence of actions (i.e., plan) that brought us into a KAB state in which the goal (that is expressed in ECQs query) is satisfied. I.e., it can be seen as a particular instance of verification problem. Hence, since verification of our proposed frameworks (i.e., GKABs and their variants) has been shown to be reducible to verification of KABs, to

implement (some of) our results, we could try to make use the available results in [78]. Concerning the specification language, to bring these theoretical results into the people in business processes management area, it might also be desirable to study the correspondence between typical business processes specification language (e.g., BPMN [148]) with our specification language. Once the correspondence is established, we can implement a translation between two formalisms. Some works that might be related to this direction can be found in [144, 145], which involve translations from Petri Nets to DCDSs.

- **Syntactic Restriction Based on Golog Structure.** The work by [24] proposes two syntactic restrictions for obtaining decidability in DCDSs. Basically, those restrictions are properties over a data flow graph that is constructed from the actions specification. Moreover, such data flow graph essentially captures all possible data flow that can be induced by the specified actions. However, since it is only constructed from the actions specification, it over estimates the possible data flow. This is the case because some actions might not be executable, or some actions might never be executed after a particular action. On the other hand, a Golog program basically captures some information about the possible data flow. Essentially, it has some information about some possible sequence of actions. Hence, it might be interesting to investigate whether we can exploit such information, get a better data flow information, and hence devise a better syntactic restriction (i.e., obtain a better decidable class).
- **Embracing Quantitative Aspects.** Another further interesting research direction is to consider the quantitative aspects of the system. One possibility is to deal with the problem of verifying whether the system under study satisfies some quantitative properties specified in a certain language. For instance we can have that each transition in the transition system can be decorated with some values which possibly representing the cost of execution the corresponding action (similar to the transition systems in the work by [48]). Then, one might be interested to check whether we can reach a particular state in such a way that the total cost of actions that are executed is less than a particular value.

BIBLIOGRAPHY

- [1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison Wesley Publ. Co., 1995. (Cited on pages 11, 26, 32, and 303.)
- [2] Serge Abiteboul, Omar Benjelloun, and Tova Milo. Positive Active XML. In *Proc. of the 23rd ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS)*, pages 35–45. ACM Press, 2004. (Cited on page 5.)
- [3] Serge Abiteboul, Omar Benjelloun, and Tova Milo. The Active XML project: an overview. *Very Large Database J.*, 17(5):1019–1040, 2008. (Cited on page 5.)
- [4] Serge Abiteboul, Luc Segoufin, and Victor Vianu. Static analysis of Active XML systems. In *Proc. of the 27th ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS)*, pages 221–230, 2008. doi: 10.1145/1376916.1376948. (Cited on page 5.)
- [5] Serge Abiteboul, Pierre Bourhis, Alban Galland, and Bogdan Marinoiu. The AXML artifact model. In *Proc. of the 16th Int. Symp. on Temporal Representation and Reasoning (TIME)*, pages 11–17, 2009. (Cited on pages 1 and 5.)
- [6] Serge Abiteboul, Luc Segoufin, and Victor Vianu. Static analysis of Active XML systems. *ACM Trans. on Database Systems*, 34(4):23:1–23:44, 2009. doi: 10.1145/1620585.1620590. (Cited on page 5.)
- [7] J. Arzac and Y. Kodratoff. Some techniques for recursion removal from recursive functions. *TOPLAS*, 4(2):295–322, 1982. (Cited on page 126.)
- [8] Alessandro Artale and Enrico Franconi. A survey of temporal extensions of description logics. *Ann. of Mathematics and Artificial Intelligence*, 1–4:171–210, 2000. (Cited on pages 76 and 304.)
- [9] Alessandro Artale, Carsten Lutz, and David Toman. A description logic of change. In *Proc. of the 19th Int. Workshop on Description Logic (DL)*, volume 189 of *CEUR Electronic Workshop Proceedings*, <http://ceur-ws.org/>, 2006. (Cited on pages 76 and 304.)
- [10] Alessandro Artale, Roman Kontchakov, Carsten Lutz, Frank Wolter, and Michael Zakharyashev. Temporalising tractable description logics. In *Proc. of the 14th Int. Symp. on Temporal Representation and Reasoning (TIME)*, pages 11–22, 2007. (Cited on pages 76 and 304.)
- [11] Alessandro Artale, Diego Calvanese, Roman Kontchakov, and Michael Zakharyashev. The *DL-Lite* family and relations. *J. of Artificial Intelligence Research*, 36:1–69, 2009. (Cited on page 27.)

- [12] Franz Baader. Logic-based knowledge representation. In *Artificial Intelligence Today*, volume 1600 of *Lecture Notes in Computer Science*, pages 13–41. Springer, 1999. (Cited on page 3.)
- [13] Franz Baader and Benjamin Zarrieß. Verification of Golog programs over description logic actions. In *Proc. of the 9th Int. Symp. on Frontiers of Combining Systems (FroCoS)*, volume 8152 of *Lecture Notes in Computer Science*, pages 181–196. Springer, September 2013. (Cited on pages 6, 7, 303, and 304.)
- [14] Franz Baader, Carsten Lutz, Maja Milicic, Ulrike Sattler, and Frank Wolter. Integrating description logics and action formalisms: First results. In *Proc. of the 20th Nat. Conf. on Artificial Intelligence (AAAI)*, 2005. (Cited on pages 6, 7, 303, and 304.)
- [15] Franz Baader, Carsten Lutz, Maja Milicic, Ulrike Sattler, and Frank Wolter. Integrating description logics and action formalisms: First results. In *Proc. of the 18th Int. Workshop on Description Logic (DL)*, number 147 in CEUR Electronic Workshop Proceedings, <http://ceur-ws.org/>, 2005. (Cited on pages 6, 7, 303, and 304.)
- [16] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2nd edition, 2007. (Cited on pages 3, 27, and 76.)
- [17] Franz Baader, Silvio Ghilardi, and Carsten Lutz. LTL over description logic axioms. In *Proc. of the 11th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR)*, pages 684–694, 2008. (Cited on pages 76 and 304.)
- [18] Franz Baader, Silvio Ghilardi, and Carsten Lutz. LTL over description logic axioms. *ACM Trans. on Computational Logic*, 13(3):21:1–21:32, 2012. (Cited on page 76.)
- [19] Franz Baader, Martin Knechtel, and Rafael Peñaloza. Context-dependent views to axioms and consequences of semantic web ontologies. *John Wiley & Sons*, 12–13:22–40, 2012. (Cited on page 171.)
- [20] Babak Bagheri Hariri, Diego Calvanese, Giuseppe De Giacomo, and Riccardo De Masellis. Verification of conjunctive-query based semantic artifacts. In *Proc. of the 24th Int. Workshop on Description Logic (DL)*, volume 745 of *CEUR Electronic Workshop Proceedings*, <http://ceur-ws.org/>, 2011. (Cited on pages 2, 6, 7, and 8.)
- [21] Babak Bagheri Hariri, Diego Calvanese, Giuseppe De Giacomo, Riccardo De Masellis, and Paolo Felli. Foundations of relational artifacts verification. In *Proc. of the 9th Int. Conf. on Business Process Management (BPM)*, volume 6896 of *Lecture Notes in Computer Science*, pages 379–395. Springer, 2011. (Cited on pages 5, 6, 7, and 77.)

- [22] Babak Bagheri Hariri, Diego Calvanese, Giuseppe De Giacomo, Riccardo De Masellis, Marco Montali, and Paolo Felli. Verification of description logic Knowledge and Action Bases. In *Proc. of the 20th Eur. Conf. on Artificial Intelligence (ECAI)*, pages 103–108, 2012. (Cited on pages 2, 7, 11, 22, 53, and 303.)
- [23] Babak Bagheri Hariri, Diego Calvanese, Giuseppe De Giacomo, Alin Deutsch, and Marco Montali. Verification of relational data-centric dynamic systems with external services. CoRR Technical Report arXiv:1203.0024, arXiv.org e-Print archive, 2012. Available at <http://arxiv.org/abs/1203.0024>. (Cited on pages 6, 7, 11, and 41.)
- [24] Babak Bagheri Hariri, Diego Calvanese, Giuseppe De Giacomo, Alin Deutsch, and Marco Montali. Verification of relational data-centric dynamic systems with external services. In *Proc. of the 32nd ACM SIGACT SIGMOD SIGAI Symp. on Principles of Database Systems (PODS)*, pages 163–174, 2013. (Cited on pages 6, 7, 8, 9, 11, 14, 17, 22, 39, 41, 47, 51, 53, 71, 72, 278, 281, 302, and 307.)
- [25] Babak Bagheri Hariri, Diego Calvanese, Marco Montali, Ario Santoso, and Dmitry Solomakhin. Verification of semantically-enhanced artifact systems (extended version). CoRR Technical Report arXiv:1308.6292, arXiv.org e-Print archive, 2013. Available at <http://arxiv.org/abs/1308.6292>. (Cited on page 272.)
- [26] Babak Bagheri Hariri, Diego Calvanese, Marco Montali, Ario Santoso, and Dmitry Solomakhin. Verification of semantically-enhanced artifact systems. In *Proc. of the 11th Int. Joint Conf. on Service Oriented Computing (ICSOC)*, volume 8274 of *Lecture Notes in Computer Science*, pages 600–607. Springer, 2013. (Cited on page 272.)
- [27] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008. (Cited on pages 2, 5, 76, and 285.)
- [28] Jorge A. Baier, Christian Fritz, and Sheila A. McIlraith. Exploiting procedural Domain Control Knowledge in state-of-the-art planners. In *Proc. of the 17th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, pages 26–33, 2007. (Cited on page 303.)
- [29] François Bancilhon and Nicolas Spyratos. Update semantics of relational views. *ACM Trans. on Database Systems*, 6(4):557–575, 1981. (Cited on page 305.)
- [30] Francesco Belardinelli, Alessio Lomuscio, and Fabio Patrizi. An abstraction technique for the verification of artifact-centric systems. In *Proc. of the 13th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR)*, pages 319–328, 2012. (Cited on pages 1, 6, 9, and 302.)
- [31] Francesco Belardinelli, Alessio Lomuscio, and Fabio Patrizi. Verification of GSM-based artifact-centric systems through finite abstraction. In *Proc. of the 10th Int. Joint Conf. on Service Oriented Computing (ICSOC)*, volume 7636 of *Lecture Notes in Computer Science*, pages 17–31. Springer, 2012. doi: 10.1007/978-3-642-34321-6_2. (Cited on pages 1, 19, and 282.)

- [32] Francesco Belardinelli, Alessio Lomuscio, and Fabio Patrizi. Verification of agent-based artifact systems. *J. of Artificial Intelligence Research*, 51:333–376, 2014. doi: 10.1613/jair.4424. (Cited on pages 6 and 302.)
- [33] Francesco Belardinelli, Davide Grossi, and Alessio Lomuscio. Finite abstractions for the verification of epistemic properties in open multi-agent systems. In *Proc. of the 24th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 854–860, 2015. (Cited on page 303.)
- [34] Daniela Berardi, Diego Calvanese, and Giuseppe De Giacomo. Reasoning on UML class diagrams. *Artificial Intelligence*, 168(1–2):70–118, 2005. (Cited on page 3.)
- [35] Leopoldo E. Bertossi. Consistent query answering in databases. *SIGMOD Record*, 35(2):68–76, 2006. (Cited on pages 8, 128, and 304.)
- [36] K. Bhattacharya, N. S. Caswell, S. Kumaran, A. Nigam, and F. Y. Wu. Artifact-centered operational modeling: Lessons from customer engagements. *IBM Systems J.*, 46(4):703–721, 2007. (Cited on page 1.)
- [37] K. Bhattacharya, C. Gerede, R. Hull, R. Liu, and J. Su. Towards formal analysis of artifact-centric business process models. In *Proc. of the 5th Int. Conf. on Business Process Management (BPM)*, volume 4714 of *Lecture Notes in Computer Science*, pages 288–234. Springer, 2007. (Cited on page 1.)
- [38] Kamal Bhattacharya, Robert Guttman, Kelly Lyman, Fenno F. Heath, Santhosh Kumaran, Prabir Nandi, Frederick Y. Wu, Prasanna Athma, Christoph Freiberg, Lars Johannsen, and Andreas Staudt. A model-driven approach to industrializing discovery processes in pharmaceutical research. *IBM Systems J.*, 44(1):145–162, 2005. (Cited on page 1.)
- [39] Meghyn Bienvenu. On the complexity of consistent query answering in the presence of simple ontologies. In *Proc. of the 26th AAAI Conf. on Artificial Intelligence (AAAI)*, pages 705–711, 2012. (Cited on page 8.)
- [40] Meghyn Bienvenu and Riccardo Rosati. Query-based comparison of OBDA specifications. In *Proc. of the 28th Int. Workshop on Description Logic (DL)*, volume 1350 of *CEUR Electronic Workshop Proceedings*, <http://ceur-ws.org/>, 2015. URL <http://ceur-ws.org/Vol-1350/paper-11.pdf>. (Cited on page 4.)
- [41] Egon Börger. Approaches to modeling business processes: a critical analysis of BPMN, workflow patterns and YAWL. *J. of Software and Systems Modeling*, 11:305–318, 2012. (Cited on page 3.)
- [42] Loris Bozzato, Chiara Ghidini, and Luciano Serafini. Comparing contextual and flat representations of knowledge: a concrete case about football data. In *Proc. of the 7th Int. Conf. on Knowledge Capture (K-CAP)*, pages 9–16. ACM Press, 2013. (Cited on page 172.)
- [43] Julian Bradfield and Colin Stirling. Modal mu-calculi. In *Handbook of Modal Logic*, volume 3, pages 721–756. Elsevier, 2007. (Cited on pages 6, 11, and 90.)

- [44] Marco Cadoli, Luigi Palopoli, and Maurizio Lenzerini. Datalog and description logics: Expressive power. Preliminary report. In *Proc. of the 9th Int. Workshop on Description Logic (DL)*, number WS-96-05 in AAAI Technical Report. AAAI Press, 1996. (Cited on page 3.)
- [45] Andrea Cali, Georg Gottlob, Thomas Lukasiewicz, Bruno Marnette, and Andreas Pieris. Datalog+/-: A family of logical knowledge representation and query languages for new applications. In *Proc. of the 25th IEEE Symp. on Logic in Computer Science (LICS)*, pages 228–242, 2010. (Cited on page 3.)
- [46] D. Calvanese, M. Giese, P. Haase, I. Horrocks, T. Hubauer, Y. Ioannidis, E. Jiménez-Ruiz, E. Kharlamov, H. Kllapi, J. Klüwer, M. Koubarakis, S. Lamparter, R. Möller, C. Neuenstadt, T. Nordtveit, Ö. Özcep, M. Rodriguez-Muro, M. Roshchin, F. Savo, M. Schmidt, A. Soylu, A. Waaler, and D. Zheleznyakov. Optique: OBDA solution for big data. In *Revised Selected Papers of ESWC 2013 Satellite Events*, volume 7955 of *Lecture Notes in Computer Science*, pages 293–295. Springer, 2013. ISBN 978-3-642-41241-7. doi: 10.1007/978-3-642-41242-4_48. (Cited on page 4.)
- [47] D. Calvanese, M. Giese, P. Haase, I. Horrocks, T. Hubauer, Y. Ioannidis, E. Jiménez-Ruiz, E. Kharlamov, H. Kllapi, J. Klüwer, M. Koubarakis, S. Lamparter, R. Möller, C. Neuenstadt, T. Nordtveit, Ö. Özcep, M. Rodriguez-Muro, M. Roshchin, M. Ruzzi, F. Savo, M. Schmidt, A. Soylu, A. Waaler, and D. Zheleznyakov. The Optique project: Towards OBDA systems for industry (Short paper). In *Proc. of the 10th Int. Workshop on OWL: Experiences and Directions (OWLED)*, volume 1080 of *CEUR Electronic Workshop Proceedings*, <http://ceur-ws.org/>, 2013. (Cited on page 4.)
- [48] Diego Calvanese and Ario Santoso. Best service synthesis in the weighted roman model. In *Proc. of the 4th Central-European Workshop on Services and their Composition (ZEUS 2012)*, volume 847 of *CEUR Electronic Workshop Proceedings*, <http://ceur-ws.org/>, pages 42–49, 2012. (Cited on page 307.)
- [49] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, and Riccardo Rosati. Ontology-based database access. In *Proc. of the 15th Ital. Conf. on Database Systems (SEBD)*, pages 324–331, 2007. (Cited on page 4.)
- [50] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Tractable reasoning and efficient query answering in description logics: The *DL-Lite* family. *J. of Automated Reasoning*, 39(3):385–429, 2007. (Cited on pages 11, 27, 38, 39, and 278.)
- [51] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. EQL-Lite: Effective first-order query processing in description logics. In *Proc. of the 20th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 274–279, 2007. (Cited on pages 11, 35, and 37.)
- [52] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Path-based identification constraints in description logics.

- In *Proc. of the 11th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR)*, pages 231–241, 2008. (Cited on pages 28, 29, 30, and 37.)
- [53] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, Mariano Rodríguez-Muro, and Riccardo Rosati. Ontologies and databases: The *DL-Lite* approach. In Sergio Tessaris and Enrico Franconi, editors, *Reasoning Web. Semantic Technologies for Informations Systems – 5th Int. Summer School Tutorial Lectures (RW)*, volume 5689 of *Lecture Notes in Computer Science*, pages 255–356. Springer, 2009. (Cited on pages 4, 17, 27, 31, 37, 271, 272, and 280.)
 - [54] Diego Calvanese, Evgeny Kharlamov, Werner Nutt, and Dmitriy Zheleznyakov. Evolution of *DL-Lite* knowledge bases. In *Proc. of the 9th Int. Semantic Web Conf. (ISWC)*, volume 6496 of *Lecture Notes in Computer Science*, pages 112–128. Springer, 2010. (Cited on pages 12, 13, 128, and 130.)
 - [55] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Actions and programs over description logic knowledge bases: A functional approach. In Gerhard Lakemeyer and Sheila A. McIlraith, editors, *Knowing, Reasoning, and Acting: Essays in Honour of Hector Levesque*. College Publications, 2011. (Cited on page 73.)
 - [56] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, Mariano Rodríguez-Muro, Riccardo Rosati, Marco Ruzzi, and Domenico Fabio Savo. The Mastro system for ontology-based data access. *Semantic Web J.*, 2(1):43–53, 2011. Listed among the 5 most cited papers in the first five years of the Semantic Web Journal. (Cited on page 4.)
 - [57] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Marco Montali, and Ario Santoso. Semantically-governed data-aware processes. In *Proc. of the 1st Int. Workshop on Knowledge-intensive Business Processes (KiBP 2012)*, volume 861 of *CEUR Electronic Workshop Proceedings*, <http://ceur-ws.org/>, pages 21–32, 2012. (Cited on page 272.)
 - [58] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Marco Montali, and Ario Santoso. Ontology-based governance of data-aware processes. In *Proc. of the 6th Int. Conf. on Web Reasoning and Rule Systems (RR)*, volume 7497 of *Lecture Notes in Computer Science*, pages 25–41. Springer, 2012. (Cited on page 272.)
 - [59] Diego Calvanese, Evgeny Kharlamov, Marco Montali, and Dmitriy Zheleznyakov. Inconsistency tolerance in OWL 2 QL knowledge and action bases - statement of interest. In *Proc. of the 9th Int. Workshop on OWL: Experiences and Directions (OWLED)*, volume 849 of *CEUR Electronic Workshop Proceedings*, <http://ceur-ws.org/>, 2012. (Cited on page 8.)
 - [60] Diego Calvanese, Babak Bagheri Hariri, Riccardo De Masellis, Domenico Lembo, Marco Montali, Ario Santoso, Dimitry Solomakhin, and Sergio Tessaris. Techniques and tools for KAB, to manage action linkage with the Artifact Layer –

- Iteration 2. Deliverable ACSI-D2.4.2, ACSI Consortium, May 2013. (Cited on pages 19, 282, and 295.)
- [61] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Data complexity of query answering in description logics. *Artificial Intelligence*, 195:335–360, 2013. doi: 10.1016/j.artint.2012.10.003. (Cited on page 135.)
 - [62] Diego Calvanese, Giuseppe De Giacomo, and Marco Montali. Foundations of data-aware process analysis: A database theory perspective. In *Proc. of the 32nd ACM SIGACT SIGMOD SIGAI Symp. on Principles of Database Systems (PODS)*, pages 1–12, 2013. (Cited on page 5.)
 - [63] Diego Calvanese, Giuseppe De Giacomo, Marco Montali, and Fabio Patrizi. Verification and synthesis in description logic based dynamic systems. In *Proc. of the 7th Int. Conf. on Web Reasoning and Rule Systems (RR)*, volume 7994 of *Lecture Notes in Computer Science*, pages 50–64. Springer, 2013. (Cited on pages 6, 9, 11, and 303.)
 - [64] Diego Calvanese, Evgeny Kharlamov, Marco Montali, Ario Santoso, and Dmitriy Zheleznyakov. Verification of inconsistency-aware knowledge and action bases (extended version). CoRR Technical Report arXiv:1304.6442, arXiv.org e-Print archive, 2013. Available at <http://arxiv.org/abs/1304.6442>. (Cited on page 128.)
 - [65] Diego Calvanese, Evgeny Kharlamov, Marco Montali, Ario Santoso, and Dmitriy Zheleznyakov. Verification of inconsistency-aware knowledge and action bases. In *Proc. of the 26th Int. Workshop on Description Logic (DL)*, volume 1014 of *CEUR Electronic Workshop Proceedings*, <http://ceur-ws.org/>, pages 107–119, 2013. (Cited on page 128.)
 - [66] Diego Calvanese, Evgeny Kharlamov, Marco Montali, Ario Santoso, and Dmitriy Zheleznyakov. Verification of inconsistency-aware knowledge and action bases. In *Proc. of the 23rd Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 810–816. AAAI Press, 2013. (Cited on page 128.)
 - [67] Diego Calvanese, İsmail İlkan Ceylan, Marco Montali, and Ario Santoso. Adding context to knowledge and action bases. In *Workshop Notes of the 6th Int. Workshop on Acquisition, Representation and Reasoning about Context with Logic (ARCOE-Logic 2014)*, volume arXiv:1412.7965 of *CoRR Technical Report*, pages 25–36. arXiv.org e-Print archive, 2014. Available at <http://arxiv.org/abs/1412.7965>. (Cited on page 172.)
 - [68] Diego Calvanese, İsmail İlkan Ceylan, Marco Montali, and Ario Santoso. Verification of context-sensitive knowledge and action bases. In *Proc. of the 14th Eur. Conf. on Logics in Artificial Intelligence (JELIA)*, volume 8761 of *Lecture Notes in Computer Science*, pages 514–528. Springer, 2014. (Cited on page 172.)
 - [69] Diego Calvanese, Davide Lanti, Martin Rezk, Mindaugas Slusnys, and Guohui Xiao. A scalable benchmark for OBDA systems: Preliminary report. In *Proc.*

- of the 3rd Int. Workshop on OWL Reasoner Evaluation (ORE), volume 1207 of *CEUR Electronic Workshop Proceedings*, <http://ceur-ws.org/>, 2014. (Cited on page 4.)
- [70] Diego Calvanese, Benjamin Cogrel, Elem Guzel Kalayci, Sarah Komla Ebri, Roman Kontchakov, Davide Lanti, Martin Rezk, Mariano Rodriguez-Muro, and Guohui Xiao. OBDA with the ontop framework. In *Proc. of the 23th Ital. Symp. on Advanced Database Systems (SEBD)*, 2015. (Cited on page 4.)
 - [71] Diego Calvanese, Giuseppe De Giacomo, Marco Montali, and Fabio Patrizi. Description logic based dynamic systems: Modeling, verification, and synthesis. In *Proc. of the 24th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 4247–4253, 2015. (Cited on page 303.)
 - [72] Diego Calvanese, Giuseppe De Giacomo, and Mikhail Soutchanski. On the undecidability of the situation calculus extended with description logic ontologies. In *Proc. of the 24th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 2840–2846, 2015. (Cited on page 76.)
 - [73] Diego Calvanese, Marco Montali, Fabio Patrizi, and Andrey Rivkin. Leveraging relational technology for data-centric dynamic systems. In *Proc. of the 23th Ital. Symp. on Advanced Database Systems (SEBD)*, 2015. (Cited on pages 6 and 306.)
 - [74] Diego Calvanese, Marco Montali, Fabio Patrizi, and Andrey Rivkin. Implementing data-centric dynamic systems over a relational dbms. In *Proc. of the 9th Alberto Mendelzon Int. Workshop on Foundations of Data Management (AMW)*, volume 1378 of *CEUR Electronic Workshop Proceedings*, <http://ceur-ws.org/>, 2015. (Cited on pages 6 and 306.)
 - [75] Diego Calvanese, Marco Montali, and Ario Santoso. Verification of generalized inconsistency-aware knowledge and action bases (extended version). CoRR Technical Report arXiv:1504.08108, arXiv.org e-Print archive, 2015. Available at <http://arxiv.org/abs/1504.08108>. (Cited on pages 73 and 128.)
 - [76] Diego Calvanese, Marco Montali, and Ario Santoso. Inconsistency management in generalized knowledge and action bases. In *Proc. of the 28th Int. Workshop on Description Logic (DL)*, volume 1350, 2015. (Cited on pages 73 and 128.)
 - [77] Diego Calvanese, Marco Montali, and Ario Santoso. Verification of generalized inconsistency-aware knowledge and action bases. In *Proc. of the 24th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 2847–2853. AAAI Press, 2015. (Cited on pages 73 and 128.)
 - [78] Diego Calvanese, Marco Montali, Fabio Patrizi, and Michele Stawowy. Plan synthesis for knowledge and action bases. In *Proc. of the 25th Int. Joint Conf. on Artificial Intelligence (IJCAI)*. AAAI Press, 2016. To appear. (Cited on pages 306 and 307.)
 - [79] Piero Cangialosi, Giuseppe De Giacomo, Riccardo De Masellis, and Riccardo Rosati. Conjunctive artifact-centric services. In *Proc. of the 8th Int. Joint*

- Conf. on Service Oriented Computing (ICSOC)*, volume 6470 of *Lecture Notes in Computer Science*, pages 318–333. Springer, 2010. (Cited on page 1.)
- [80] İsmail İlkan Ceylan and Rafael Peñaloza. The Bayesian description logic \mathcal{BEL} . In *Proc. of the 7th Int. Joint Conf. on Automated Reasoning (IJCAR)*, volume 8562 of *Lecture Notes in Computer Science*, pages 480–494. Springer, 2014. (Cited on pages 171 and 175.)
- [81] Tian Chao, David Cohn, Adrian Flatgard, Sandy Hahn, Mark Linehan, Prabir Nandi, Anil Nigam, Florian Pinel, John Vergo, and Frederick y Wu. Artifact-based transformation of IBM Global Financing, a case study. In *Proc. of the 7th Int. Conf. on Business Process Management (BPM)*, *Lecture Notes in Computer Science*. Springer, 2009. (Cited on page 1.)
- [82] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996. (Cited on page 2.)
- [83] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. The MIT Press, Cambridge, MA, USA, 1999. (Cited on pages 11 and 39.)
- [84] Jens Claßen and Gerhard Lakemeyer. A logic for non-terminating Golog programs. In *Proc. of the 11th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR)*, pages 589–599, 2008. (Cited on pages 122 and 303.)
- [85] David Cohn and Richard Hull. Business artifacts: A data-centric approach to modeling business operations and processes. *Bull. of the IEEE Computer Society Technical Committee on Data Engineering*, 32(3):3–9, 2009. (Cited on pages 1, 5, 41, and 77.)
- [86] Mads Dam. CTL* and ECTL* as fragments of the modal μ -calculus. In *Proc. of the 17th Colloquium on Trees in Algebra and Programming (CAAP’92)*, volume 581 of *Lecture Notes in Computer Science*, pages 145–164. Springer, 1992. (Cited on page 284.)
- [87] Elio Damaggio, Richard Hull, and Roman Vaculín. On the equivalence of incremental and fixpoint semantics for business artifacts with Guard-Stage-Milestone lifecycles. *Information Systems*, 38(4):561–584, 2013. (Cited on pages 6, 9, 19, 282, and 283.)
- [88] Umeshwar Dayal and Philip A. Bernstein. On the correct translation of update operations on relational views. *ACM Trans. on Database Systems*, 7(3):381–416, September 1982. ISSN 0362-5915. (Cited on page 305.)
- [89] Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in the situation calculus. In *Proc. of the 15th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 1221–1226, 1997. (Cited on page 122.)

- [90] Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1–2):109–169, 2000. (Cited on pages 121 and 122.)
- [91] Giuseppe De Giacomo, Yves Lesperance, and Fabio Patrizi. Bounded Situation Calculus action theories and decidable verification. In *Proc. of the 13th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR)*, pages 467–477, 2012. (Cited on page 302.)
- [92] A. de Medeiros, C. Pedrinaci, W. van der Aalst, J. Domingue, M. Song, A. Rozinat, B. Norton, and L. Cabral. An outlook on semantic business process mining and monitoring. In *Proc. of On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops*, volume 4806 of *Lecture Notes in Computer Science*, pages 1244–1255. Springer, 2007. (Cited on page 4.)
- [93] Alin Deutsch, Richard Hull, Fabio Patrizi, and Victor Vianu. Automatic verification of data-centric business processes. In *Proc. of the 12th Int. Conf. on Database Theory (ICDT)*, pages 252–267, 2009. (Cited on pages 5, 8, 9, and 77.)
- [94] Marlon Dumas. Integrated data and process management: Finally? In *Proc. of the 1st Int. Workshop on Knowledge-intensive Business Processes (KiBP)*, volume 861 of *CEUR Electronic Workshop Proceedings*, <http://ceur-ws.org/>, pages 21–32, 2012. (Cited on page 1.)
- [95] Marc Ehrig, Agnes Koschmider, and Andreas Oberweis. Measuring similarity between semantic business process models. In *Proc. of the fourth Asia-Pacific conference on Conceptual modelling - Volume 67*, pages 71–80. Australian Computer Society, Inc., 2007. (Cited on page 4.)
- [96] Thomas Eiter and Georg Gottlob. On the complexity of propositional knowledge base revision, updates and counterfactuals. *Artificial Intelligence*, 57:227–270, 1992. (Cited on page 128.)
- [97] Ramez A. ElMasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison Wesley Publ. Co., 5th edition, 2007. (Cited on pages 3, 5, 26, and 32.)
- [98] E. Allen Emerson. Model checking and the Mu-calculus. In N. Immerman and P. Kolaitis, editors, *Proceedings of the DIMACS Symposium on Descriptive Complexity and Finite Models*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 185–214. American Mathematical Society Press, 1996. ISBN 0-8218-0517-7. (Cited on page 284.)
- [99] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: Semantics and query answering. In *Proc. of the 9th Int. Conf. on Database Theory (ICDT)*, pages 207–224, 2003. (Cited on pages 6 and 7.)
- [100] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: Semantics and query answering. *Theoretical Computer Science*, 336(1):89–124, 2005. (Cited on page 72.)

- [101] Richard Fikes and Tom Kehler. The role of frame-based representation in reasoning. *Communications of the ACM*, 28(9):904–920, 1985. (Cited on page 3.)
- [102] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3):189 – 208, 1971. ISSN 0004-3702. (Cited on page 59.)
- [103] Giorgos Flouris, Dimitris Manakanatas, Haridimos Kondylakis, Dimitris Plexousakis, and Grigoris Antoniou. Ontology change: Classification and survey. *Knowledge Engineering Review*, 23(2):117–152, 2008. (Cited on page 128.)
- [104] Martin Fowler and Kendall Scott. *UML Distilled – Applying the Standard Object Modeling Language*. Addison Wesley Publ. Co., 1997. (Cited on page 3.)
- [105] Enrico Franconi, Alessandro Mosca, and Dmitry Solomakhin. ORM2 encoding into description logics. In *Proc. of the 25th Int. Workshop on Description Logic (DL)*, volume 846 of *CEUR Electronic Workshop Proceedings*, <http://ceur-ws.org/>, 2012. (Cited on page 3.)
- [106] Christian Fritz, Jorge A. Baier, and Sheila A. McIlraith. ConGolog, Sin Trans: Compiling ConGolog into Basic Action Theories for planning and beyond. In *Proc. of the 11th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR)*, pages 600–610, 2008. (Cited on pages 126 and 303.)
- [107] C. E. Gerede and J. Su. Specification and verification of artifact behaviors in business process models. In *Proc. of the 5th Int. Conf. on Service Oriented Computing (ICSOC)*, 2007. (Cited on pages 1, 5, and 9.)
- [108] C. E. Gerede, K. Bhattacharya, and J. Su. Static analysis of business artifact-centric operational models. In *Proc. of the 5th Int. Conf. on Service Oriented Computing (ICSOC)*, 2007. (Cited on pages 1, 5, and 9.)
- [109] Vasilis C Gerogiannis, Achilles D Kameas, and Panayotis E Pintelas. Comparative study and categorization of high-level petri nets. *J. of Systems and Software*, 43(2):133–160, 1998. (Cited on page 3.)
- [110] Giuseppe De Giacomo, Yves Lespérance, Fabio Patrizi, and Stavros Vassos. Progression and verification of situation calculus agents with bounded beliefs. In *Proc. of the 13th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 141–148, 2014. (Cited on page 302.)
- [111] Giuseppe De Giacomo, Yves Lespérance, Fabio Patrizi, and Stavros Vassos. LTL verification of online executions with sensing in bounded situation calculus. In *Proc. of the 21st Eur. Conf. on Artificial Intelligence (ECAI)*, pages 369–374, 2014. (Cited on page 303.)
- [112] Fausto Giunchiglia and Paolo Bouquet. Introduction to contextual reasoning. an artificial intelligence perspective. In *Perspectives on Cognitive Science*, pages 138–159. NBU Press, 1997. (Cited on page 172.)

- [113] Pavel Gonzales, Andreas Griesmayer, and Alessio Lomuscio. Model checking tool for artifact interoperations (MOCAI) – Iteration 3. Deliverable ACSI-D2.2.3, ACSI Consortium, May 2013. (Cited on pages [282](#) and [290](#).)
- [114] Pavel Gonzalez, Andreas Griesmayer, and Alessio Lomuscio. Verifying GSM-based business artifacts. In *Proc. of the 19th IEEE Int. Conf. on Web Services (ICWS)*, pages 25–32, 2012. (Cited on page [282](#).)
- [115] Pavel Gonzalez, Andreas Griesmayer, and Alessio Lomuscio. Model checking GSM-based multi-agent systems. In *Proc. of Service-Oriented Computing - IC-SOC 2013 Workshops*, volume 8377 of *Lecture Notes in Computer Science*, pages 54 – 68. Springer, 2014. (Cited on page [6](#).)
- [116] Georg Gottlob, Paolo Paolini, and Roberto Zicari. Properties and update semantics of consistent views. *ACM Trans. on Database Systems*, 13(4):486–524, October 1988. ISSN 0362-5915. (Cited on page [305](#).)
- [117] Peter Haase, Ian Horrocks, Dag Hovland, Thomas Hubauer, Ernesto Jiménez-Ruiz, Evgeny Kharlamov, Johan W. Klüwer, Christoph Pinkel, Riccardo Rosati, Valerio Santarelli, Ahmet Soylu, and Dmitriy Zheleznyakov. Optique system: Towards ontology and mapping management in OBDA solutions. In *Proc. of the 2nd Int. Workshop on Debugging Ontologies and Ontology Mappings (WoDOOM)*, pages 21–32, 2013. (Cited on page [4](#).)
- [118] Terry Halpin. Object-role modeling: Principles and benefits. *Int. J. of Information System Modeling and Design*, 1(1):33–57, January 2010. ISSN 1947-8186. (Cited on page [3](#).)
- [119] Terry Halpin. *Object-Role Modeling Fundamentals*. Technics Publications, 2015. (Cited on page [3](#).)
- [120] Terry A. Halpin and Anthony C. Bloesch. Data modeling in UML and ORM: A comparison. *J. of Database Management*, 10(4):4–13, 1999. (Cited on page [3](#).)
- [121] Babak Bagheri Hariri, Diego Calvanese, Marco Montali, Giuseppe De Giacomo, Riccardo De Masellis, and Paolo Felli. Description logic Knowledge and Action Bases. *J. of Artificial Intelligence Research*, 46:651–686, 2013. ISSN 1076-9757. doi: 10.1613/jair.3826. (Cited on pages [2](#), [6](#), [7](#), [8](#), [9](#), [11](#), [22](#), [39](#), [53](#), [59](#), [61](#), [72](#), [76](#), [301](#), and [303](#).)
- [122] Peter G. Harrison and Hessam Khoshnevisan. A new approach to recursion removal. *Theoretical Computer Science*, 93(1):91 – 113, 1992. (Cited on page [126](#).)
- [123] Martin Hepp, Frank Leymann, John Domingue, Alexander Wahler, and Dieter Fensel. Semantic business process management: a vision towards using semantic web services for business process management. In *Proc. of IEEE International Conference on e-Business Engineering (ICEBE 2005)*, pages 535–540, Oct 2005. (Cited on page [4](#).)
- [124] Stijn Heymans, Li Ma, Darko Anicic, Zhilei Ma, Nathalie Steinmetz, Yue Pan, Jing Mei, Achille Fokoue, Aditya Kalyanpur, Aaron Kershenbaum, Edith Schonberg, Kavitha Srinivas, Cristina Feier, Graham Hench, Branimir Wetzstein, and

- Uwe Keller. Ontology reasoning with large data repositories. In Martin Hepp, Pieter De Leenheer, Aldo de Moor, and York Sure, editors, *Ontology Management, Semantic Web, Semantic Web Services, and Business Applications*, volume 7 of *Semantic Web And Beyond Computing for Human Experience*, pages 89–128. Springer, 2008. (Cited on page 4.)
- [125] Richard Hull. Artifact-centric business process models: Brief survey of research results and challenges. In *Proc. of the 7th Int. Conf. on Ontologies, DataBases, and Applications of Semantics (ODBASE)*, volume 5332 of *Lecture Notes in Computer Science*, pages 1152–1163. Springer, 2008. doi:10.1007/978-3-540-88873-4_17. (Cited on pages 1, 2, 5, 41, 77, and 283.)
- [126] Richard Hull, Elio Damaggio, Riccardo De Masellis, Fabiana Fournier, Manmohan Gupta, Fenno Terry Heath, III, Stacy Hobson, Mark Linehan, Sridhar Maradugu, Anil Nigam, Piwadee Noi Sukaviriya, and Roman Vaculin. Business artifacts with Guard-Stage-Milestone lifecycles: Managing artifact interactions with conditions and events. In *Proc. of the 5th ACM Int. Conf. on Distributed Event-Based Systems (DEBS 2011)*, pages 51–62, 2011. (Cited on pages 6, 9, 19, 282, and 283.)
- [127] Marwane El Kharbili, Sebastian Stein, Ivan Markovic, and Elke Pulvermüller. Towards a framework for semantic business process compliance management. In *Proc. of the 1st Int. Workshop on Governance, Risk and Compliance - Applications in Information Systems (GRCIS)*, volume 339 of *CEUR Electronic Workshop Proceedings*, <http://ceur-ws.org/>, 2008. (Cited on page 4.)
- [128] Davide Lanti, Martin Rezk, Guohui Xiao, and Diego Calvanese. The NPD benchmark: Reality check for OBDA systems. In *Proc. of the 18th Int. Conf. on Extending Database Technology (EDBT)*, 2015. (Cited on page 4.)
- [129] Freddy Lécué and Alain Léger. A formal model for semantic web service composition. In *Proc. of the 5th Int. Semantic Web Conf. (ISWC)*, volume 4273 of *Lecture Notes in Computer Science*, pages 385–398. Springer, 2006. (Cited on page 3.)
- [130] Fritz Lehmann, editor. *Semantic Networks in Artificial Intelligence*. Pergamon Press, Oxford (United Kingdom), 1992. (Cited on page 3.)
- [131] Domenico Lembo and Marco Ruzzi. Consistent query answering over description logic ontologies. In *Proc. of the 1st Int. Conf. on Web Reasoning and Rule Systems (RR)*, 2007. (Cited on page 8.)
- [132] Domenico Lembo, Maurizio Lenzerini, Riccardo Rosati, Marco Ruzzi, and Domenico Fabio Savo. Inconsistency-tolerant semantics for description logics. In *Proc. of the 4th Int. Conf. on Web Reasoning and Rule Systems (RR)*, pages 103–117, 2010. (Cited on pages 12, 13, 128, and 304.)
- [133] Domenico Lembo, Maurizio Lenzerini, Riccardo Rosati, Marco Ruzzi, and Domenico Fabio Savo. Query rewriting for inconsistent *DL-Lite* ontologies. In *Proc. of the 5th Int. Conf. on Web Reasoning and Rule Systems (RR)*, 2011. (Cited on pages 12 and 128.)

- [134] H. J. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. Scherl. GOLOG: A logic programming language for dynamic domains. *J. of Logic Programming*, 31:59–84, 1997. (Cited on pages 7, 11, 73, 121, 122, 301, and 303.)
- [135] Hector J. Levesque. Foundations of a functional approach to knowledge representation. *Artificial Intelligence*, 23:155–212, 1984. (Cited on pages 7 and 76.)
- [136] Yanhong A. Liu and Scott D. Stoller. From recursion to iteration: What are the optimizations? In *Proc. of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 73–82. ACM Press, 1999. (Cited on page 126.)
- [137] Ruopeng Lu and Shazia Sadiq. A survey of comparative business process modeling approaches. In *Proc. of 10th Int. Conf. on Business Information Systems (BIS)*, volume 4439 of *Lecture Notes in Computer Science*, pages 82–94. Springer, 2007. (Cited on page 3.)
- [138] Carsten Lutz, Frank Wolter, and Michael Zakharyashev. Temporal description logics: A survey. In *Proc. of the 15th Int. Symp. on Temporal Representation and Reasoning (TIME)*, pages 3–14, 2008. (Cited on pages 76 and 304.)
- [139] Carsten Lutz, David Toman, and Frank Wolter. Conjunctive query answering in the description logic \mathcal{EL} using a relational database system. In *Proc. of the 21st Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 2070–2075, 2009. (Cited on page 4.)
- [140] John McCarthy. Notes on formalizing context. In *Proc. of the 13th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 555–560, 1993. (Cited on page 172.)
- [141] Sheila A. McIlraith, Tran Cao Son, and Honglei Zeng. Semantic web services. *IEEE Intelligent Systems*, 16(2):46 – 53, Mar/Apr 2001. (Cited on page 3.)
- [142] Brahim Medjahed, Athman Bouguettaya, and Ahmed K. Elmagarmid. Composing web services on the semantic web. *Very Large Database J.*, 2003. (Cited on page 3.)
- [143] Andreas Meyer, Sergey Smirnov, and Mathias Weske. *Data in business processes*. Universitätsverlag Potsdam, 2011. (Cited on page 1.)
- [144] Marco Montali and Andrey Rivkin. Formal verification of petri nets with names. In *Proc. of the 11th Int. Workshop on Web Services and Formal Methods (WS-FM)*, Lecture Notes in Computer Science. Springer, 2014. (Cited on page 307.)
- [145] Marco Montali and Andrey Rivkin. Model checking petri nets with names using data-centric dynamic systems. *Formal Aspects of Computing*, 2016. To appear. (Cited on page 307.)
- [146] Marco Montali, Diego Calvanese, and Giuseppe De Giacomo. Verification of data-aware commitment-based multiagent systems. In *Proc. of the 13th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 157–164, 2014. ISBN 978-1-4503-2738-1. (Cited on pages 6, 7, 9, 11, 53, 62, and 303.)

- [147] A. Nigam and N. S. Caswell. Business artifacts: An approach to operational specification. *IBM Systems J.*, 42(3):428–445, 2003. (Cited on pages 1, 41, and 283.)
- [148] Object Management Group (OMG). Business process model and notation (BPMN). Technical report, Object Management Group (OMG), January 2014. Available at <http://www.omg.org/spec/BPMN/2.0.2/>. (Cited on pages 3 and 307.)
- [149] David Michael Ritchie Park. Finiteness is Mu-ineffable. *Theoretical Computer Science*, 3(2):173–181, 1976. (Cited on pages 39 and 186.)
- [150] Benjamin C. Pierce. Linguistic foundations for bidirectional transformations: invited tutorial. In *Proc. of the 31st ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS)*, pages 61–64. ACM Press, 2012. (Cited on page 305.)
- [151] Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Linking data to ontologies. *J. on Data Semantics*, X:133–173, 2008. doi: 10.1007/978-3-540-77688-8_5. (Cited on pages 4, 27, 271, 272, 275, 278, 280, and 281.)
- [152] Antonella Poggi, Mariano Rodriguez-Muro, and Marco Ruzzi. Ontology-based database access with DIG-Mastro and the OBDA Plugin for Protégé. In Kendall Clark and Peter F. Patel-Schneider, editors, *Proc. of the 4th Int. Workshop on OWL: Experiences and Directions (OWLED DC)*, 2008. (Cited on page 4.)
- [153] Jan Recker, Michael Rosemann, Marta Indulska, and Peter Green. Business process modeling - a comparative analysis. *J. of the Association of Information Systems*, 10(4), 2009. (Cited on pages 3 and 171.)
- [154] Manfred Reichert. Process and data: Two sides of the same coin? In *Proc. of the On the Move Confederated Int. Conf. (OTM 2012)*, volume 7565 of *Lecture Notes in Computer Science*, pages 2–19. Springer, 2012. doi: 10.1007/978-3-642-33606-5_2. (Cited on page 1.)
- [155] Raymond Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press, 2001. (Cited on page 302.)
- [156] Clay Richardson. Warning: Don’t assume your business processes use master data. In *Proc. of the 8th Int. Conf. on Business Process Management (BPM)*, volume 6336 of *Lecture Notes in Computer Science*, pages 11–12. Springer, 2010. (Cited on page 1.)
- [157] Mariano Rodriguez-Muro and Diego Calvanese. Towards an open framework for ontology based data access with Protégé and DIG 1.1. In *Proc. of the 5th Int. Workshop on OWL: Experiences and Directions (OWLED)*, volume 432 of *CEUR Electronic Workshop Proceedings*, <http://ceur-ws.org/>, 2008. (Cited on page 4.)

- [158] Mariano Rodriguez-Muro and Diego Calvanese. Dependencies: Making ontology based data access work in practice. In *Proc. of the 5th Alberto Mendelzon Int. Workshop on Foundations of Data Management (AMW)*, volume 749 of *CEUR Electronic Workshop Proceedings*, <http://ceur-ws.org/>, 2011. (Cited on page 4.)
- [159] Mariano Rodriguez-Muro and Diego Calvanese. Dependencies to optimize ontology based data access. In *Proc. of the 24th Int. Workshop on Description Logic (DL)*, volume 745 of *CEUR Electronic Workshop Proceedings*, <http://ceur-ws.org/>, 2011. (Cited on page 4.)
- [160] Mariano Rodriguez-Muro and Diego Calvanese. High performance query answering over *DL-Lite* ontologies. In *Proc. of the 13th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR)*, pages 308–318, 2012. (Cited on pages 4 and 271.)
- [161] Mariano Rodriguez-Muro, Lina Lubyte, and Diego Calvanese. Realizing ontology based data access: A plug-in for Protégé. In *Proc. of the ICDE Workshop on Information Integration Methods, Architectures, and Systems (IIMAS 2008)*, pages 286–289. IEEE Computer Society Press, 2008. (Cited on page 4.)
- [162] Michael Rosemann, Jan Recker, and Christian Flender. Contextualisation of business processes. *Int. J. of Business Process Integration and Management*, 3 (1):47–60, 2008. (Cited on page 171.)
- [163] Alessandro Russo, Massimo Mecella, Fabio Patrizi, and Marco Montali. Implementing and running data-centric dynamic systems. In *Proc. of the 7th IEEE Int. Conf. on Service Oriented Computing and Applications (SOCA)*, pages 225–232. IEEE, 2013. (Cited on pages 6 and 306.)
- [164] Ario Santoso. When data, knowledge and processes meet together. In *Proc. of the 6th Int. Conf. on Web Reasoning and Rule Systems (RR)*, volume 7497 of *Lecture Notes in Computer Science*, pages 291–296. Springer, 2012. (Cited on page 272.)
- [165] Juan F. Sequeda, Marcelo Arenas, and Daniel P. Miranker. OBDA: Query rewriting or materialization? In practice, both! In *Proc. of the 13th Int. Semantic Web Conf. (ISWC)*, volume 8796 of *Lecture Notes in Computer Science*, pages 535–551. Springer, 2014. (Cited on page 4.)
- [166] Luciano Serafini and Martin Homola. Contextualized knowledge repositories for the semantic web. *J. of Web Semantics*, 12:64–87, 2012. (Cited on page 172.)
- [167] R. M. Smullyan. *First Order Logic*. Springer, Berlin (Germany), 1968. (Cited on page 25.)
- [168] John F. Sowa, editor. *Principles of Semantic Networks: Explorations in the Representation of Knowledge*. Morgan Kaufmann, 1991. (Cited on page 3.)
- [169] Ahmet Soyly, Evgeny Kharlamov, Dmitriy Zheleznyakov, Ernesto Jiménez-Ruiz, Martin Giese, and Ian Horrocks. OptiqueVQS: Visual query formulation for

- OBDA. In *Proc. of the 27th Int. Workshop on Description Logic (DL)*, volume 1193 of *CEUR Electronic Workshop Proceedings*, <http://ceur-ws.org/>, pages 725–728, 2014. URL http://ceur-ws.org/Vol-1193/paper_88.pdf. (Cited on page 4.)
- [170] Colin Stirling. *Modal and Temporal Properties of Processes*. Springer, 2001. (Cited on pages 39, 67, 68, 91, 93, and 186.)
- [171] A. Swartz. MusicBrainz: a semantic web service. *IEEE Intelligent Systems*, 17(1):76 – 77, Jan/Feb 2002. (Cited on page 3.)
- [172] OASIS Web Services Business Process Execution Language (WSBP EL) TC. Web services business process execution language version 2.0. Technical report, OASIS, April 2007. Available at <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html>. (Cited on page 3.)
- [173] David Toribio Gomez, Catherine Murphy O Connor, Pieter De Leenheer, Pierre Malarme, Jordi De Vos, Fabiana Fournier, David Boaz, Boudewijn van Dongen, Dirk Fahland, Massimiliano de Leoni, and Marlon Dumas. Energy and FRIS use case definition and requirements. Deliverable ACSI-D5.1, ACSI Consortium, March 2010. (Cited on page 293.)
- [174] David Toribio Gomez, Catherine Murphy O Connor, Pieter De Leenheer, Pierre Malarme, Jordi De Vos, Stijn Christiaens, Fabiana Fournier, and Lior Limonad. Energy and FRIS “as-is” assessment. Deliverable ACSI-D5.2, ACSI Consortium, May 2011. (Cited on page 293.)
- [175] David Toribio Gómez, Catherine Murphy-O Connor, Pieter De Leenheer, and Pierre Malarme. Deployment and evaluation of pilots using the ACSI Hub System – Iteration 1. Deliverable ACSI-D5.3, ACSI Consortium, June 2012. (Cited on page 293.)
- [176] David Toribio Gómez, Catherine Murphy-O Connor, Pieter De Leenheer, and Pierre Malarme. Deployment and evaluation of pilots using the ACSI Hub System – Results and evaluation. Deliverable ACSI-D5.5, ACSI Consortium, May 2013. (Cited on page 293.)
- [177] UML. Unified Modeling Language (UML) superstructure, version 2.0. Available at <http://www.uml.org/>, August 2005. (Cited on page 3.)
- [178] Wil MP van der Aalst. Business process management: A comprehensive survey. *ISRN Software Engineering*, 2013, 2013. (Cited on page 4.)
- [179] W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: yet another workflow language. *Information Systems*, 30(4):245 – 275, 2005. ISSN 0306-4379. (Cited on page 3.)
- [180] Mathias Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer, 2007. (Cited on page 4.)

- [181] Branimir Wetzstein, Zhilei Ma, Agata Filipowska, Monika Kaczmarek, Sami Bhiri, Silvestre Losada, Jose-Manuel Lopez-Cobo, and Laurent Cicurel. Semantic business process management: A lifecycle based requirements analysis. In *Proc. of the Workshop on Semantic Business Process and Product Lifecycle Management (SBPM)*, volume 251 of *CEUR Electronic Workshop Proceedings*, <http://ceur-ws.org/>, 2007. (Cited on page 4.)
- [182] Marianne Winslett. *Updating Logical Databases*. Cambridge University Press, 1990. (Cited on page 128.)
- [183] Frank Wolter and Michael Zakharyashev. Temporalizing description logics. In M. de Rijke and D. Gabbay, editors, *Proc. of the 2th Int. Workshop on Frontiers of Combining Systems (FroCoS)*, Amsterdam, 1999. Wiley. (Cited on pages 76 and 304.)
- [184] William A. Woods. Understanding subsumption and taxonomy: A framework for progress. In J. F. Sowa, editor, *Principles of Semantic Networks*, pages 45–94. Morgan Kaufmann, 1991. (Cited on page 3.)
- [185] Benjamin Zarrieß and Jens Claßen. Verifying CTL properties of GOLOG programs over local-effect actions. In *Proc. of the 21st Eur. Conf. on Artificial Intelligence (ECAI)*, 2014. (Cited on page 122.)
- [186] Benjamin Zarrieß and Jens Claßen. Verification of knowledge-based programs over description logic actions. In *Proc. of the 24th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 3278–3284, 2015. (Cited on pages 6, 7, 303, and 304.)

*“There comes a time
when you have to choose
between turning the page
and closing the book.”*

–Josh Jameson

Thanks for reading.

*“A great article might be meaningless
without the readers.”*